Learn JavaFX Game and App Development

With FXGL 17

Dr. Almas Baimagambetov

Apress®

Learn JavaFX Game and App Development

With FXGL 17

Dr. Almas Baimagambetov

Apress®

Learn JavaFX Game and App Development: With FXGL 17

Dr. Almas Baimagambetov
Brighton, East Sussex, UK

ISBN-13 (pbk): 978-1-4842-8624-1 ISBN-13 (electronic): 978-1-4842-8625-8
https://doi.org/10.1007/978-1-4842-8625-8

Copyright © 2022 by Dr. Almas Baimagambetov

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Development Editor: Spandana Chatterjee
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-8624-1. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Dr. Almas Baimagambetov is a principal lecturer in computer science at the University of Brighton, UK. He has considerable software development experience and is a huge fan of open source. His prominent contributions to the JVM community on GitHub include the FXGL game engine, collaborations on numerous JavaFX projects, a wide range of open source games, and a collection of practical tutorials. Almas also has a YouTube channel focused on Java, Kotlin, JavaFX, Unity, and Unreal Engine.

About the Technical Reviewer

Simon Jackson is a long-time software engineer and architect with many years of Unity game development experience, as well as an author of several Unity game development titles. He loves to both create Unity projects as well as lend a hand to help educate others, whether it's via a blog, vlog, user group, or major speaking event.

His primary focus at the moment is with the XRTK (Mixed Reality Toolkit) project, which is aimed at building a cross-platform Mixed Reality framework to enable both VR and AR developers to build efficient solutions in Unity and then build/distribute them to as many platforms as possible.

CHAPTER 1

Introduction

JavaFX is a modern high-performance graphical user interface (GUI) toolkit for Java. It provides hardware acceleration support on a range of platforms, including desktop, mobile, and embedded, allowing the development of cross-platform applications. However, to develop games with JavaFX effectively, numerous domain-specific concepts are needed. To address this need, the FXGL game engine extends JavaFX and brings support for real-world game development techniques (examples seen in Figure 1-1). These include the entity-component model, pathfinding, physics, particle systems, sprite sheet animations, and many other features. As a result, JavaFX developers can produce games more quickly and more effectively with FXGL and easily package their games to native platform images, including Android and iOS.

© Dr. Almas Baimagambetov 2022
A. Baimagambetov, *Learn JavaFX Game and App Development*,
https://doi.org/10.1007/978-1-4842-8625-8_1

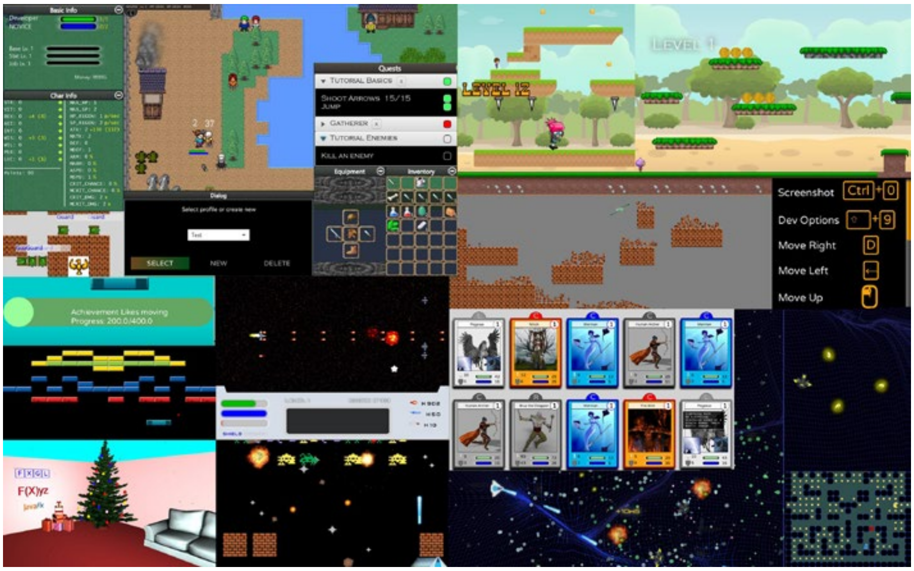


***Figure 1-1.** A compilation of screenshots from games developed with FXGL*

The FXGL game engine is fully open source. This allows any FXGL user to contribute new features or bug fixes back to the project. The project is solely supported by the community users, whether individuals or companies. At the time of writing this book, all development team members work on this project in their own time. The project source code can be found at `https://github.com/AlmasB/FXGL`.

The FXGL project initially started off as a collection of code snippets used to help deliver lectures on game development courses at the University of Brighton and help with developing tutorials for YouTube. As the code base grew, it made sense to develop it into a publicly accessible game engine in Java. As such, FXGL is not meant to compete with existing engines, but instead its goal is to provide Java users with an alternative tool to develop high-performance applications. During the early stages of development, the following software projects played an influential role in the engine's architecture and implementation: Unity, Unreal Engine, libGDX, jMonkey, CRYENGINE, X-Ray, OpenJK, Duality, and Phaser.

The underlying infrastructure and the rendering engine are provided by the JavaFX GUI toolkit. JavaFX is a modern alternative to Swing for building rich client applications in Java. It is developed in collaboration with many individuals and companies, constantly improving its presence on a wide range of supported platforms. Building on this support enables FXGL to develop truly cross-platform games with numerous game-specific features.

The features provided by FXGL can be put into several categories: internal framework services, game logic, artificial intelligence (AI), physics, gameplay, input, user interface (UI), graphics, audio, file input output (IO), networking, hardware access, and external tool integration. These categories can be described as follows:

- Internal framework services provide access to the engine timer and life cycle, allowing to listen for specific events and handle them appropriately from code.
- In FXGL, game logic is implemented as a combination of the Entity-Component model, which defines game objects, and the game world, which acts as a container for game objects. As with any popular data structure, the game world allows querying and filtering data (game objects) to suit developer needs.
- The AI category mostly comprises two areas: navigation and behavior. The navigation system allows game objects to find their way around the game level without passing through obstacles, whereas the behavior logic gives the game objects some level of artificial intelligence.

- The category of physics can also be split into two: collision detection and rigid body simulation. The former tells us if two game objects are overlapping, which in turn can be used for various gameplay mechanics, such as picking up a health kit. The latter, also known as rigid body dynamics, can be used to simulate a range of physics properties for game object interaction, such as friction, elasticity, and density.
- In terms of gameplay, FXGL provides facilities for achievements, in-game notifications, quick-time events, as well as built-in mini-games.
- The user input is handled via an intuitive set of bindings between user actions (such as shoot or jump) and triggers (such as the spacebar key or the left mouse button). The same API (set of functions accessible from code) can be used to mock physical triggers when running on mobile.
- The user interface aspect of the engine is built on top of JavaFX and enhances its visual controls. For example, FXGL provides in-game menus, overlays, camera animations, and 9-slice UI generation.
- The graphics features of FXGL are also built on the ones provided by JavaFX since the engine uses JavaFX's rendering pipeline. The improvements include a highly customizable particle system, runtime texture manipulation, and editable custom 3D shapes. An example application with some built-in 3D shapes can be seen in Figure 1-2.

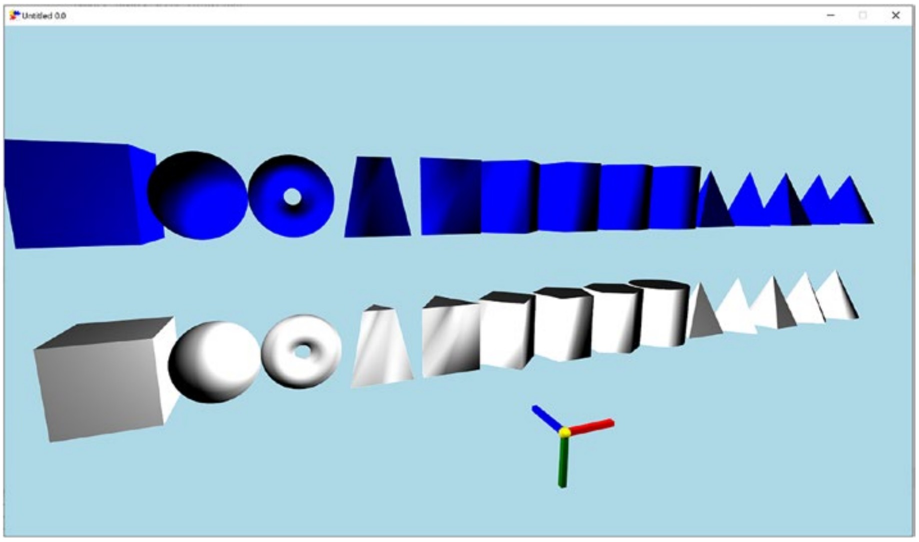


Figure 1-2. *An example of a 3D scene in FXGL*

- The audio service allows developers to play short sound effects and long background music files. It also controls certain audio properties, such as volume.
- All of the IO calls are wrapped by the file system service, which simplifies writing to and reading from the local storage (whether it is on desktop or mobile).
- The networking service is implemented by extending the standard JDK (Java Development Kit) capabilities. For example, the high-level interface does not distinguish between TCP and UDP connections and provides a unified way to handle incoming and outgoing messages.
- The hardware support includes Xbox and PlayStation controllers and a range of API available from the Gluon Attach project, which provides access to mobile hardware capabilities.

- Finally, the following external tools are supported: Tiled map editor to build game world levels and FXGL dialogue editor for easily generating complex dialogues between in-game characters.

For completeness, it should be noted that these features are not exclusive to FXGL. There are other alternatives available when entering the world of Java game development. In particular, popular frameworks and engines include libGDX and jMonkeyEngine. Both of these projects have been developed for a longer period of time, compared to FXGL. These projects provide a large number of mature game development tools to Java game developers. The FXGL project is not meant to compete with these technologies; instead, the aim is to provide an alternative approach to Java game development, particularly to JavaFX developers, who are already familiar with the JavaFX API set.

This book is designed to be suitable for both beginners and experts in Java and/or JavaFX who wish to develop apps and games with FXGL while improving Java and/or JavaFX skills. After completing this book, the reader will be able to

- Make use of advanced Java and JavaFX concepts
- Explain and implement real-world game development concepts in a general-purpose programming language, such as Java
- Develop professional cross-platform, including desktop and mobile, applications, and games using the FXGL game engine
- Design and implement scalable gameplay-specific functionality involving physics, artificial intelligence, and graphics systems in FXGL

- Deploy JavaFX and FXGL applications and games using platform-specific native images

The high-level overview of the book is as follows:

- In Chapter 2, we cover requisite concepts from Java and JavaFX that will be used throughout this book. We will also introduce the reader to the FXGL game engine and its development workflow.
- We start our game development journey in Chapter 3, where we develop an arcade game similar to Pong. Through this process, we will learn about the Entity-Component model used in FXGL to create a powerful abstraction of the game world.
- We build on this in chapter 4, where we develop a platformer game using the physics engine and learn about collision detection.
- An important concept of game AI is covered in Chapter 5 by developing a maze action game like Pac-Man.
- Many visually complex features related to graphics and rendering will be discussed in Chapter 6. In this chapter, we also cover UI elements and the animation system in FXGL.
- Chapter 7 is focused on general-purpose (nongame) applications that can be developed using FXGL. These include business applications, graphical editors, and visualization tools. The chapter also covers the packaging of JavaFX and FXGL applications, including games, to later deploy to end users.

- Finally, in Chapter 8, we conclude and discuss details of future projects that can be built on the contents of this book.

Each chapter incrementally builds on the previous chapters and introduces a new topic. The code snippets provided in this book are either in Java or pseudocode that is similar to Java. The examples given use the following software versions:

- Java 17,
- JavaFX 17
- FXGL 17
- IntelliJ 2021

An example setup project using these technologies is provided in the next chapter. The reader is encouraged to use the same or newer versions. However, it should be noted that newer versions may not have exactly the same workflow process and there may be some tweaking required. On this note, let us begin our journey with FXGL 17.

CHAPTER 2

Requisite Concepts

In this chapter, we will cover fundamental concepts from Java, JavaFX, and FXGL. This knowledge will be required to understand the contents of this book. More specifically, we will look at

- Java theory – classes and methods
- How to use IntelliJ and Maven to add external dependencies
- How to create a simple JavaFX application with basic user interface controls
- How to create a simple FXGL application

Readers who are familiar with these topics, may skip this chapter or parts of it.

Java Basics

This section will introduce (or recap for those already familiar with the concepts) fundamental Java basics, including classes and methods. We will consider how to create classes, instance-level (nonstatic) variables, and class-level (static) variables. We will also recap how to add nonstatic and static methods to a class. Finally, we will briefly talk about inheritance and the `@Override` annotation.

© Dr. Almas Baimagambetov 2022
A. Baimagambetov, *Learn JavaFX Game and App Development*,
https://doi.org/10.1007/978-1-4842-8625-8_2

Classes

In Java, and many other programming languages, classes are templates for objects. Classes are used to model or represent some abstract concepts, such as a collision detection handler, or real-world things, such as a chair. Defining a class is straightforward:

```
class Chair { }
```

The preceding code will typically be stored in a file called Chair.java, where the name of the file matches the class name. It should be noted that per the Java language naming convention, classes start with an uppercase letter. For each new word in the class name, the word also starts with an uppercase letter. So if we had a class for a different type of chair, such as "wooden chair," then the name would be `WoodenChair`.

Creating an instance of a class is known as instantiating that class:

```
Chair diningChair = new Chair();
```

In the preceding code, we created an instance, called `diningChair`, of a class called `Chair`. The instance (`diningChair`) is also called an object. An object is simply an instance of any class. Classes can have variables, which store information. When adding a variable to a class definition, each instance of that class will have its own copy of these variables that can be manipulated. For example, we can add a variable of type int (recall that an int, or integer, represents a whole number):

```
class Chair {
    int cost;
}
```

The variable cost could store information related to how much a single chair costs:

```
diningChair.cost = 50;
```

So chair diningChair costs 50 units of currency. We can create another object from the same class and assign a different cost value:

```
Chair officeChair = new Chair();
officeChair.cost = 75;
```

Note that although objects diningChair and officeChair are both of type Chair, they are two distinct objects: chair diningChair costs 50, and chair officeChair costs 75. If we change values of some variables in diningChair, the values of variables in officeChair will not change. This is one of the crucial concepts of object-oriented programming: each object has its own properties.

We have just talked about variables that belong to an object. Such variables that only affect the instance to which they belong are known as instance level. In the preceding example, the variable cost is an instance-level variable. In addition to instance-level variables, we can also define class-level variables, or static variables. As implied by name, they belong to a class, rather than to an object. Consider the following:

```
class Chair {
    static int quality;
    int cost;
}
```

We have just created a static variable of type int, named quality, which belongs to the class Chair and not to an individual object of type Chair. Since quality is a class-level variable, we typically access it via the class, not the object:

```
Chair.quality = 3;
```

We will note that all objects that access the value of quality will read the same value of 3, since there is only one static variable. Having covered the basics of the "class" concept, we now move on to methods.

Methods

As explained earlier, variables allow us to store information on a per-object, or per-class, basis. Methods allow us to define actions that can be performed, again, on a per-object, or per-class, basis. We will continue with our `Chair` class and add a simple method to it, called move, in Listing 2-1.

Listing 2-1. An example of adding a method to a class

```
class Chair {
    static int quality;
    int cost;

    void move(int x, int y) {
        // some code
    }
}
```

Methods (known as functions or procedures in some languages) are blocks of code that can be reused. A method definition has four important parts:

- **Name:** The name of our method is "move". Typically, the name is a meaningful action since methods define actions – what an object can do. By convention, they start with a lowercase letter, then each new word in the name is capitalized.
- **Return type:** The return type, in our case, is void, which is a special keyword, and it means that the method does not return any value. Any type (e.g., int) can be used instead of "void" to signify that the method returns a value of that type. Returning a value from a method is a common way to extract some result from the method. However, in our example, there is no meaningful result we expect; hence, we use "void".

- **List of parameters:** The list of parameters shows the types that the method expects when being called. In the preceding example, there are two expected values of type int: x and y. Using parameters is a common way for developers to pass some information into a method.
- **Body:** Finally, the body of a method is a block of code that is located between the curly braces "{" and "}" of that method. In our example, we added the "// some code" placeholder instead of the method body for clarification purposes. The method body is essentially the code that will be executed when the method is invoked (that is when the method is called).

As with variables, which can be static, methods can also be prefixed with the static keyword, in which case they belong to a class. Suppose we wish to have an action that allows us to determine the material of chair objects produced by calling `new Chair()`. Assuming that all chairs in our abstract model have the same material (of some type named `Material`), there would be no difference between the material of `c1` and the material of `c2`. If this is the case, we can simply add a static method, called `getMaterial()`, to the `Chair` class, in Listing 2-2.

Listing 2-2. An example of a static method

```
class Chair {
    static int quality;
    int cost;

    static Material getMaterial() {
        // some code
    }
    void move(int x, int y) {
        // some code
    }
}
```

Similar to how a static variable is accessed, that is, via the class, static methods are also accessed in this way. So to call the newly created method, we would write `Chair.getMaterial()`, rather than `c1.getMaterial()`. If, further down the line, we decide that each chair can have its own material, then we can remove the static keyword from the method. Then the method becomes nonstatic, and we would need to call it via `c1.getMaterial()` or `c2.getMaterial()` as before.

Sometimes, it is useful to define classes that have similar, yet distinct, types. For example, consider a dining room chair and an office chair. Both of these things are of type chair, and they do have some common functionalities, yet the full set of their functionalities is not the same. A trivial example would be that an office chair has wheels but a dining chair does not. If we were to create a separate class for both of these types of chairs, for example:

```
class DiningChair
class OfficeChair
```

then we would readily see that we would need to duplicate quite a lot of code. Clearly, this is not an ideal solution. To avoid this, Java, as a language, provides the inheritance feature. In simple terms, we can define a “supertype” called Chair where we can keep all of the common functionalities that a chair would have, and then we can define “subtypes” (or subclasses) `DiningChair` and `OfficeChair` that “extend” from Chair and only contain code that is relevant to them. Thus, with this approach, we can provide a nice and clean solution to our little problem of duplicated code. To extend from a class, one can write as follows:

```
class DiningChair extends Chair {}
class OfficeChair extends Chair {}
```

In some cases, it is possible for two (or more) subtypes to share functionality (think methods) but in a way where each subtype performs the functionality in its own way. For this concept, let us pick a different, not related to chairs, example. A commonly used one in many textbooks is the class `Animal` with the method `speak()`.

```
class Animal {
    void speak() { }
}
```

Suppose we have two subtypes of Animal: Cat and Dog. Both types can speak, but in their own way. So while the functionality remains the same, the way it is performed is different for each subtype of Animal.

To implement this functionality in Java, we have the `@Override` annotation, which can be added to a method to signify that it overrides the method with the same name from the supertype class. Let us consider a concrete example in Listing 2-3. The first thing to note is the `@Override` annotation on the `speak()` method. It tells the developer that `speak()` in the class `Cat` overrides the method `speak()` in the class `Animal`. So when the user calls `speak()`, if the object on which the call was made is of type `Cat`, then the `speak()` method that lives inside the `Cat` class will be called and not the one from the class `Animal`. We will use "overrides" in a few places when building FXGL games; hence, this concept will become more important as we continue through this book. The reader will be reminded of its usage when we will get there.

Listing 2-3. An example of using the @Override annotation

```
class Dog extends Animal {
    @Override
    void speak() {
        // bark
    }
}
```

```
class Cat extends Animal {
    @Override
    void speak() {
        // meow
    }
}
```

Having recapped Java basics, we will now consider how to add libraries to Java projects. Libraries are reusable collections of pre-built code that are a common way to solve known problems. They are an important tool in any developer's arsenal.

Adding External Libraries

In this section, we will see how to use the IntelliJ IDE and the built-in Maven support to add an external library to a Java project. We will also download any prerequisite software. If you do not yet have an installation of JDK 17 on your system, the general-availability JDK 17 release can be downloaded from `https://jdk.java.net/17/` for all major OS platforms: Windows, macOS, and Linux. All you need to do is download the appropriate ".zip" file and extract it into a location of choice.

In Java, there are several ways to add an external library to your project. You can use Maven, Gradle, or a stand-alone ".jar" file. We will consider a common approach using Maven, which is a dependency management (or build) tool. Build tools are responsible for downloading dependencies (such as JavaFX and FXGL) a software project has so that the developer does not need to waste their time manually installing these external libraries. Modern development tools, such as the IntelliJ editor (available from `www.jetbrains.com/idea/`), already come with Maven support. To simplify our task, we will use this support and create a simple project with Maven.

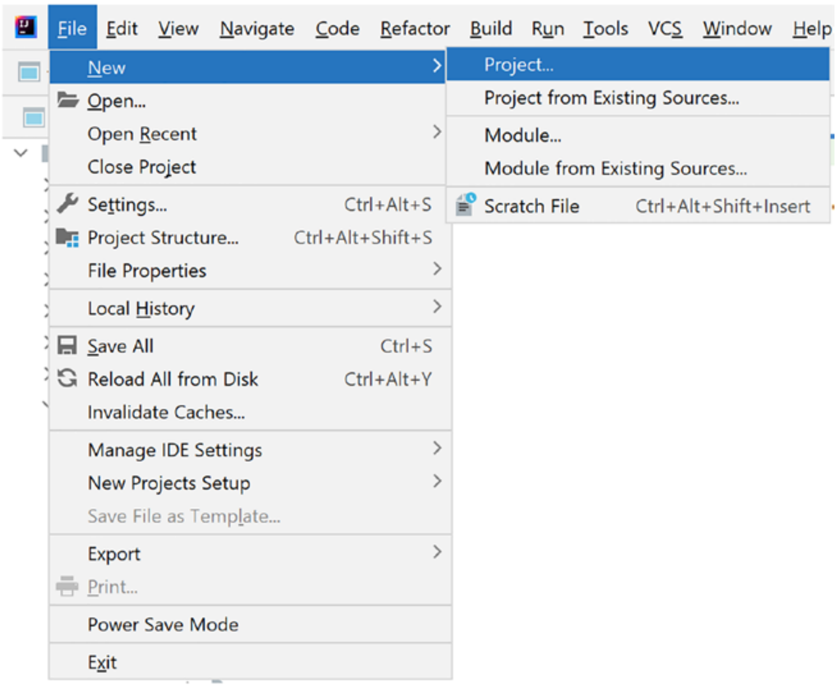
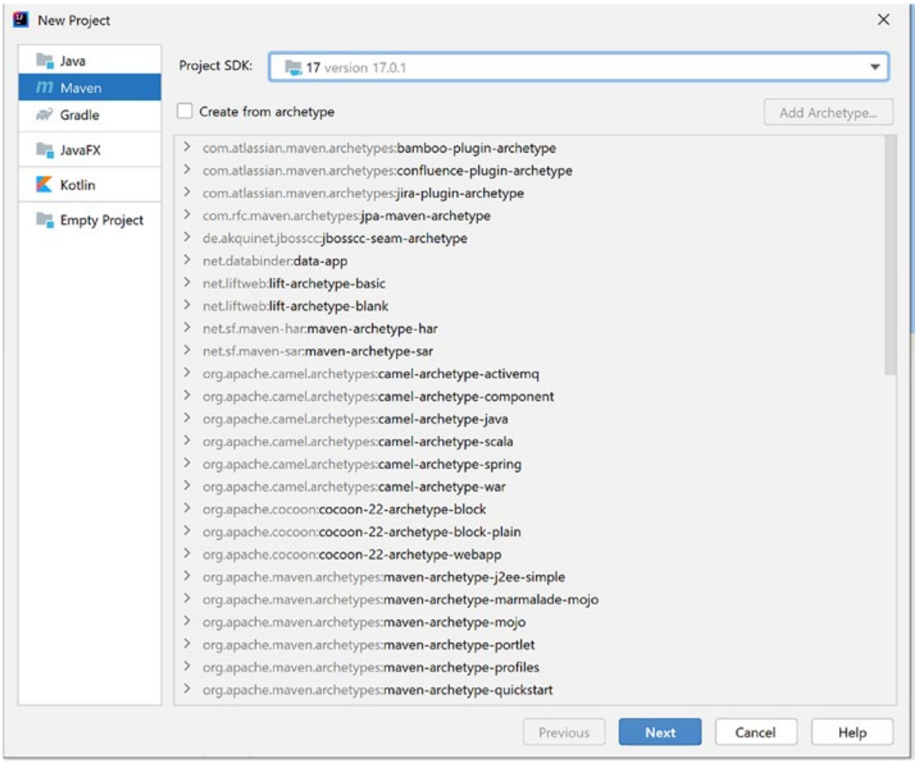


Figure 2-1. *The File ➤ New ➤ Project menu in IntelliJ IDE 2021.2.2 (Community Edition)*

Open up IntelliJ and proceed with the menu options shown in Figure 2-1. Specifically, the options are File ➤ New ➤ Project. This sequence will lead you to the following window in Figure 2-2. The first thing to ensure is that your project SDK says 17 (or higher). In Figure 2-2, the project SDK (close to the top of the image) says version 17.0.1, which is JDK 17. If you have a different version selected, then you can click on the drop-down menu and select the appropriate version or click "Add JDK", and then navigate to your JDK installation folder that you downloaded from the link that was given previously. Now that you have a suitable JDK selected, we will select the Maven project on the left-hand side (see Figure 2-2) and then click Next. In the next window, we can give the name to our project, such as JavaFXApp (or BasicFXGLGame) and choose the directory in which the source code will be stored.



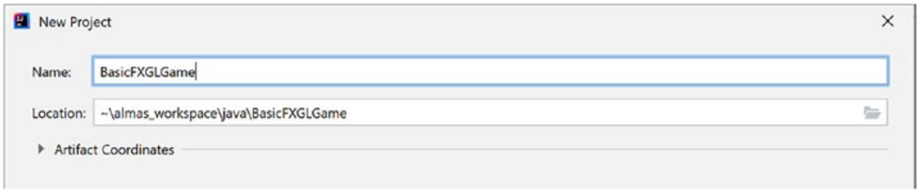
Project SDK: 17 version 17.0.1

Create from archetype

- com.atlassian.maven.archetypes:bamboo-plugin-archetype
- com.atlassian.maven.archetypes:confluence-plugin-archetype
- com.atlassian.maven.archetypes:jira-plugin-archetype
- com.rfc.maven.archetypes:jpa-maven-archetype
- de.akquinet.jbosscc:jbosscc-seam-archetype
- net.databinder:data-app
- net.liftweb:lift-archetype-basic
- net.liftweb:lift-archetype-blank
- net.sf.maven-har:maven-archetype-har
- net.sf.maven-sar:maven-archetype-sar
- org.apache.camel.archetypes:camel-archetype-activemq
- org.apache.camel.archetypes:camel-archetype-component
- org.apache.camel.archetypes:camel-archetype-java
- org.apache.camel.archetypes:camel-archetype-scala
- org.apache.camel.archetypes:camel-archetype-spring
- org.apache.camel.archetypes:camel-archetype-war
- org.apache.cocoon:cocoon-22-archetype-block
- org.apache.cocoon:cocoon-22-archetype-block-plain
- org.apache.cocoon:cocoon-22-archetype-webapp
- org.apache.maven.archetypes:maven-archetype-j2ee-simple
- org.apache.maven.archetypes:maven-archetype-marmalade-mojo
- org.apache.maven.archetypes:maven-archetype-mojo
- org.apache.maven.archetypes:maven-archetype-portlet
- org.apache.maven.archetypes:maven-archetype-profiles
- org.apache.maven.archetypes:maven-archetype-quickstart

Figure 2-2. *The New Project wizard window in IntelliJ IDE*

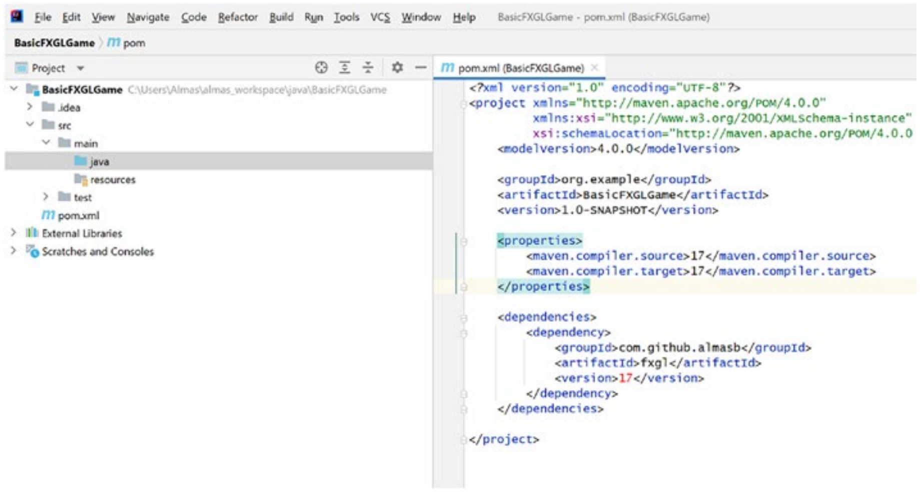
Figure 2-3 shows how the Maven project details can be modified. Once you have picked the name for the project, you can leave the other bits unchanged from the default. Click Finish. IntelliJ should now produce a project with the view that looks similar to Figure 2-4, but without the FXGL dependency. The xml file that you can see on the right is called the Project Object Model file, or pom.xml for short. This file contains the information necessary to build the project. More advanced pom files may also contain extra details related to how to manage project resources, test produced builds, generate project documentation, and even deploy the resulting artifacts.



Name: BasicFXGLGame

Location: ~\almas_workspace\java\BasicFXGLGame

Figure 2-3. *The New Maven Project window in IntelliJ*


File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
BasicFXGLGame - pom.xml (BasicFXGLGame)
BasicFXGLGame
pom
Project
BasicFXGLGame C:\Users\Almas\almas_workspace\java\BasicFXGLGame
.idea
src
main
java
resources
test
pom.xml
External Libraries
Scratches and Consoles
pom.xml (BasicFXGLGame)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>BasicFXGLGame</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.github.almasb</groupId>
            <artifactId>fxgl</artifactId>
            <version>17</version>
        </dependency>
    </dependencies>

</project>
```

Figure 2-4. *The Maven project with the generated Project Object Model (pom.xml) and an extra FXGL dependency*

We will now add the FXGL dependency (which includes JavaFX) to the project by adding a new "dependency" item inside the pom.xml file. A dependency has three parts to it: group ID, artifact name, and version. These details are given in Figure 2-4 and Listing 2-4.

Once you have added the dependency as shown in Figure 2-4, click the "refresh" icon in the top-right corner of IntelliJ IDE and the FXGL dependency will be automatically downloaded and available for use.

Listing 2-4. Adding the FXGL dependency inside the xml file

```
<dependencies>
    <dependency>
        <groupId>com.github.almasb</groupId>
        <artifactId>fxgl</artifactId>
        <version>17</version>
    </dependency>
</dependencies>
```

This step completes the addition of external libraries to Java projects using IntelliJ and Maven. We will now proceed with building some sample applications using JavaFX and FXGL.

JavaFX Basics: Creating a Simple Application

In this section, we will build a simple JavaFX application that contains commonly used controls. We start by considering a minimal JavaFX application, which would look something like the one in Listing 2-5.

Listing 2-5. A minimal JavaFX application

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class FXApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        stage.setScene(new Scene(new Pane(), 800, 600));
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

To run this application, using IntelliJ, first, create a new Java class called `FXApp` inside the `src/main/java` directory in your project – this directory is highlighted in Figure 2-4. You can now type the code in Listing 2-5 into the newly created FXApp.java file. Finally, right-click on this file and select "run" to see this simple JavaFX app running.

There are important key points to note in this source code:

- Any JavaFX application will have a class that extends Application. It is common for this class to also contain the entry point to the program – public static void main(String[] args), which simply calls launch(args). This is how most JavaFX and FXGL applications start.
- Another important code snippet is the start() method that overrides a method with the same name from the Application class. It is a necessary override and allows the developer to configure their JavaFX application during the starting phase. In our case, we do two things here: set the scene to the stage object and show the stage. The Stage class is an abstraction over an OS-specific window, which will be covered in more detail in Chapter 6 when we discuss graphics.

We will now cover the basics, including user input, interaction, and output. To do this, we extend the previous minimal example with the stand-alone example in Listing 2-6, which gives a general idea of how to implement these three basic concepts.

Listing 2-6. An example of a basic JavaFX application

```
import javafx.application.Application;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class BasicApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        stage.setScene(new Scene(createContent()));
        stage.show();
    }

    private Parent createContent() {
        TextField input = new TextField();
        Text output = new Text();

        Button button = new Button("Press");
        button.setOnAction(e -> {
            output.setText(input.getText());
        });

        VBox root = new VBox(input, button, output);
        root.setPrefSize(600, 600);
        return root;
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

The major addition in Listing 2-6 is the `createContent()` method – other bits are reasonably straightforward. We will examine this method in detail:

1. First, we create the TextField which we use to take input from the user.
2. Next, we create the Text object, which will serve as output for some information back to the user.
3. After the output, we create a button with name "Press" and provide some code to be executed when the button is pressed. Specifically, this code will set the input text data to the output text.
4. Finally, we create the root layout container that defines how the controls (input, button, output) are to be placed on the screen. In this case, we use VBox, which lays out all controls placed inside it vertically. We also set the preferred size to this layout container so that the window is automatically adjusted to facilitate this size. The root container is then returned from the method to be set as the root node of the scene.

If you run the application (as before, right-click on the Java file and click Run), the result should be similar to the screenshot in Figure 2-5. As we can observe, the layout of items is arranged in a vertical fashion.

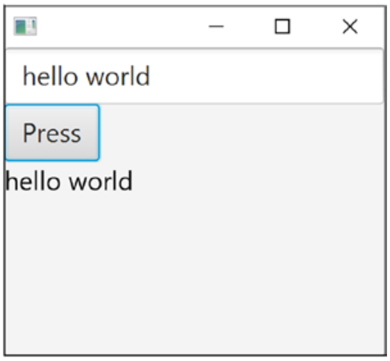


Figure 2-5. *A basic JavaFX application with three fundamental controls: TextField, Button, and Text objects*

The last major feature to consider is the JavaFX properties and how we can bind one property to another. Properties, in JavaFX, act as wrappers around values. The values can be of any type. Some commonly used types are int, double, boolean, String, and generic Object. The main advantage of using properties against already-familiar int, for example, is the fact that bound properties automatically update. Suppose we have a `TextField` that asks the user to enter their name and a Text object that simply copies the name that the user enters. Without properties, we would have to listen for each key event on the `TextField` and then manually update the Text object whenever the user entered or deleted a character:

```
text.setText(textField.getText());
```

With properties, the preceding functionality can be achieved in a neat way, which is only called once without the need to write code for any listeners.

```
text.textProperty().bind(textField.textProperty());
```

This completes the general overview of basic JavaFX features that will be used later in the book. We will now focus on building a simple FXGL application.

FXGL Basics: Creating a Simple Application

In this section, we will build a simple FXGL application, which will be our first game example. Before we proceed, let us define some requirements for our simple game:

1. The game will have a 600×600 window.
2. There is a player on the screen, represented by a blue rectangle.
3. The player can be controlled by pressing W, S, A, and D on the keyboard.
4. The user interface (UI) is represented by a single of text, which shows how many pixels the player has moved.
5. When the player moves, the UI text will update itself accordingly.

At the end of this section, you should have a game similar to what you can see in Figure 2-6. Although this may look far from a real game, its development will help you understand the basic features present in FXGL. As a result, after completing this section, you will be able to build a variety of simple games.

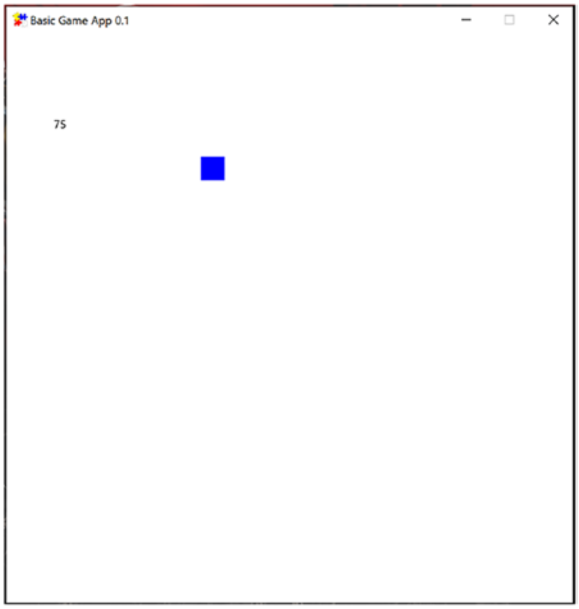


Figure 2-6. *A basic FXGL application*

The first step is to prepare our development environment. From the previous section, you should have a clear idea of how to create a new project and add FXGL as a dependency; hence, we will start immediately from creating a Java package for our game in `src/main/java`. For simplicity, let us call the package "tutorial". In that package, we are going to create a Java class named "BasicGameApp". In JavaFX applications, it is common to append "App" to the class where the `main()` method is located. This allows other developers to easily identify where the main entry point to the application is. To make your next steps a lot easier, we can open the BasicGameApp class and add the commonly used FXGL imports given in Listing 2-7.

Listing 2-7. Common FXGL imports

```
import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.dsl.FXGL;
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.input.Input;
import com.almasb.fxgl.input.UserAction;
import javafx.scene.input.KeyCode;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import java.util.Map;
```

We are now in a position to start writing some code. To start using FXGL, your "App" class needs to extend `GameApplication` and override the `initSettings()` method:

```
public class BasicGameApp extends GameApplication {
    @Override
    protected void initSettings(GameSettings settings) {}
}
```

As soon as you add the extends clause, your IDE is likely to prompt automatic generation of the overridden method. In the following subsections, we will cover other commonly used overridden methods.

Via the `initSettings()` method and the `settings` object, you can change a plethora of game settings and configure the application in a way that fits your needs. However, at the moment, we simply want to have a working application as quickly as possible. Hence, we will leave this method alone for the time being. The next method we will add is

```
public static void main(String[] args) {
    launch(args);
}
```

We already covered a simple JavaFX application in an earlier section, so this method is nothing new. This is the way to start a normal JavaFX application. Since FXGL is seamlessly integrated with JavaFX, this is also the way to start FXGL applications. As you progress through this book, you will realize that FXGL is just JavaFX with game development features. At this point, your application is already runnable, so you may wish to start the game (as per usual by right-clicking and selecting "Run"), see the results, and finally close it.

Requirement 1: Opening a Window

We will now go through each requirement we defined earlier, in turn, to develop our game. The first requirement was to open a 600×600 window, in which our game will run. As we mentioned earlier, in order to change settings, we can use the `initSettings()` method, shown in Listing 2-8.

Listing 2-8. The initSettings() method allows providing game-specific settings

```
@Override
protected void initSettings(GameSettings settings) {
    settings.setWidth(600);
    settings.setHeight(600);
    settings.setTitle("Basic Game App");
    settings.setVersion("0.1");
}
```

Using the settings object, we changed the width and height to 600. We also set the title and version of the game. Once they are set, the settings cannot be changed during runtime. This does not mean you cannot resize the application for instance; instead, this means the provided values are the startup settings to be used. You can now click "run" in your IDE, which should start the game with a 600×600 window and "Basic Game App" as the window title. We have now completed the first requirement. You can see this requirement was straightforward to achieve. You will note that other requirements are not more complex either. This is the overall aim of FXGL: to make JavaFX game development significantly easier and more streamlined.

Requirement 2: Adding the Player

The second requirement is to add a player object to our game. We are going to do this in `initGame()`. In short, this is where you set up everything that needs to be ready before the game starts. The code is given in Listing 2-9.

Listing 2-9. The initGame() method allows setting up the game and its game objects

```
private Entity player;

@Override
protected void initGame() {
    player = FXGL.entityBuilder()
            .at(300, 300)
            .view(new Rectangle(25, 25, Color.BLUE))
            .buildAndAttach();
}
```

We will cover it slowly, method by method.

- There is an instance-level field named player of type Entity. An entity is essentially a game object, as defined by FXGL. This is everything you need to know about it at this stage. We will discuss entities to a greater length in the next chapter. FXGL.entityBuilder() is the preferred way to build entities.

- Next, by calling .at(), we can set the position of the entity we are creating. In this case, it's x = 300 and y = 300. Note that a position of any entity is at its top-left point, as in JavaFX.
- Then, we tell the entity builder to create the view of our entity. In particular, we wish to use the Rectangle object, which comes directly from JavaFX, so nothing new here. We construct the rectangle by providing its width = 25, height = 25, and color. An important thing to note here is that you can use any JavaFX Node instead of the rectangle to pass into the view method and therefore provide the visual representation of your entity.
- Finally, we call .buildAndAttach(). By calling build, we can obtain the reference to the entity that we were building. As for the attach part, it conveniently allows us to add the built entity straight to the game world. We will cover the game world and what it is in the next chapter. For now, just consider it as a place where entities exist and live.

If you run the game now, you should see a blue rectangle near the center of the window. This step completes requirement 2.

Requirement 3: Adding Input

Requirement 3 states that we should be able to control the player entity using the keyboard. We will now proceed with user input. In FXGL, typically all of the input-related code is placed inside the `initInput()` method. The code in Listing 2-10 shows how to add an input action. Most of the input in FXGL is handled in this format. The developer would specify an action (some code to be run) and then bind it to a trigger (a mouse button, a key press, or some other way to provide input, such as a controller).

Listing 2-10. The initInput() method allows binding user input to in-game actions

```
@Override
protected void initInput() {
    FXGL.onKey(KeyCode.D, () -> {
        player.translateX(5); // move right 5 pixels
    });
}
```

Let us go through Listing 2-10 line by line. By calling the `onKey()` method from FXGL, we are effectively saying "while a key is being pressed, perform an action." In this case, the key is `KeyCode.D`. Again, if you have used JavaFX before, then you will know that these are exactly the same key codes used in standard event handlers. As you have noticed, to use most of the FXGL functionality, all you need to do is call `FXGL.***`, and your IDE will show all possible functions with autocomplete.

Recall from the requirements that the action we wish to preform is to move the player. So when D is pressed, we want to move the player to the right. We call `player.translateX(5)`, which moves the entity's X coordinate by 5 pixels. Note that "translate" is a terminology used in computer graphics (and JavaFX) and means move. You will note that the FXGL API is quite concise. This concise API is commonly called DSL – domain-specific language in the context of FXGL. This can be shortened even further if we use static imports. You can statically import FXGL calls by adding the following import line:

```
import static com.almasb.fxgl.dsl.FXGL.*;
```

Now that we are familiar with input handling for one key, the rest of the code should be straightforward and is given in Listing 2-11.

Listing 2-11. The initInput() method implementation for all four directions

```
@Override
protected void initInput() {
    FXGL.onKey(KeyCode.D, () -> {
        player.translateX(5); // move right 5 pixels
    });

    FXGL.onKey(KeyCode.A, () -> {
        player.translateX(-5); // move left 5 pixels
    });

    FXGL.onKey(KeyCode.W, () -> {
        player.translateY(-5); // move up 5 pixels
    });

    FXGL.onKey(KeyCode.S, () -> {
        player.translateY(5); // move down 5 pixels
    });
}
```

With the `initInput()` method populated, requirement 3 is now complete.

Requirement 4: Adding UI

The next requirement is the UI, which we handle in the `initUI()` method, given in Listing 2-12.

Listing 2-12. The initUI() method allows adding user interface objects on the screen

```
@Override
protected void initUI() {
    Text textPixels = new Text();
    textPixels.setTranslateX(50); // x = 50
    textPixels.setTranslateY(100); // y = 100

    FXGL.getGameScene().addUINode(textPixels); // add to
    the screen
}
```

In Listing 2-12, the Text class is used directly from JavaFX. In fact, for most UI objects, we simply use the JavaFX controls, since there is no need to reinvent the wheel. You should note that when we added the entity to the world, the game scene picked up the fact the entity had a view associated with it. Hence, the game scene magically added the entity to the scene graph. With UI objects, we are responsible for their additions to the scene graph, and we do exactly that by calling `getGameScene().addUINode()`. We will handle the UI updates, such as the number of pixels moved by the player a bit later on.

Requirement 5: Updating UI

In order to complete the last requirement, we are going to use a game variable. In FXGL, a game variable can be accessed and modified from any part of the game. In a sense, it is a global variable with the scope being tied to the FXGL game instance. In addition, such variables can be bound to (like with JavaFX properties). We start by creating such a variable:

```
@Override
protected void initGameVars(Map<String, Object> vars) {
    vars.put("pixelsMoved", 0);
}
```

The preceding code creates a game variable named "pixelsMoved" with the initial value of 0. Note that FXGL will automatically infer the type of the variable based on the initial value, which in this case is an integer. For example, if we used 0.0, then the type of the game variable would be a double. Next, we need to update the variable when the player moves. We can do this in the input handling section of the code.

```
FXGL.onKey(KeyCode.D, () -> {
    player.translateX(5); // move right 5 pixels
    FXGL.inc("pixelsMoved", +5);
});
```

You can do the same with the rest of the movement actions, that is, left, up, and down. The last step is to bind our UI text object we created earlier to the variable `pixelsMoved`. Ultimately, this allows FXGL to automatically update the UI based on the `pixelsMoved` value so that we don't need to update the UI manually. In `initUI()`, once we have created the `textPixels` object, we can do the following:

```
textPixels.textProperty().bind(FXGL.getWorldProperties().
intProperty("pixelsMoved").asString());
```

To expand on this, we are taking the text property of the `textPixels` UI object and binding it to the `pixelsMoved` value, which first needs to be converted to a String since it is an integer. This is the last change we are making to this example, and you now should have a simple FXGL game.

There are many areas that we have not covered yet, for example, asset loading, physics, AI, and graphics – these and other features will be discussed as we continue our journey through this book.

Summary

In this chapter, we covered requisite concepts that are used throughout this book. We recapped Java basics with respect to classes and methods. We saw how one could create a class to model an abstract or a real-world object and how methods can provide functionalities to these classes. Next, we used IntelliJ to add the JavaFX (via FXGL) external library via the built-in Maven support, which simplifies the process. Subsequently, we used the JavaFX library to create a simple application with three UI controls: text output, text input, and button. Finally, using the FXGL library as an external dependency, we implemented a simple game demo. The demo has shown us how to use input and create entities, which are game objects, and how to interact with JavaFX UI controls. In the next chapter, we will learn more about entities and how they allow us to create complex game worlds.

CHAPTER 3

Entity-Component Case Study: Develop Pong

We start our game development journey in this chapter, where we develop a clone of the Pong arcade game. We will start building your knowledge about the game development process with FXGL. Through this process, we will learn about

- Entities and how they represent game objects
- Components and how they add data and behavior to entities
- Game world and how to perform queries to access specific entities in the world

Game objects form the fundamental concepts of any game. The bigger the game, the more robust implementation of game objects should be in order to scale properly. Developers with extensive background in object-oriented programming may fall into a trap where inheritance ends up being intertwined with game object functionality, leading to a less scalable software architecture. Inheritance favors vertical hierarchy and is

© Dr. Almas Baimagambetov 2022
A. Baimagambetov, *Learn JavaFX Game and App Development*,
https://doi.org/10.1007/978-1-4842-8625-8_3

extremely beneficial in a range of use cases, including small to medium-sized games. However, the Entity-Component model, which tends to favor horizontal hierarchy, offers its own benefits and can be more suitable in large game projects. The benefits of this model will become clearer as you continue reading through this chapter. We will begin by defining what an entity is.

Entity

Any game object that you can think of (e.g., player, enemy, coin, bullet, power-up, wall, dirt, weapon, treasure box, and many others) is an entity. By itself, an entity is nothing but a generic object. Without adding components, it is almost impossible to distinguish one entity from another. Components are the crucial objects that carry data and behavior with them to define each entity in its unique way. We will expand upon this in the next sections.

In FXGL, there are a number of useful methods you can invoke on an entity.

These include

- addComponent(Component c)/ removeComponent(Component c) – Allows adding new and removing existing components.
- setProperty(String key, Object value) – Allows adding a variable of any type to the entity. These variables are useful if you do not wish to create a new component to store data. The values can be obtained by calling getInt(String key), getString(String key), getObject(String key), and other similar methods.
- getComponents() – Retrieves the list of all components attached to an entity.

- getPosition()/setPosition(Point2D p) – Returns the position/sets the position of the entity in 2D space. More specifically, the method asks the TransformComponent to provide/modify this data (as we know, entities themselves do not store any data, only their components do).
- translate(Point2D vector) – Moves the entity by the given vector. More specifically, the method asks the TransformComponent to modify position data.
- isVisible()/setVisible(boolean b) – Checks if the entity is being drawn/if set to false, the entity will not be drawn.
- getType()/setType(Object type) – Allows retrieving/setting the entity type. Typically, the enum class values are used as types.

Now that we are more familiar with what an entity is and what methods we can invoke on it, we will consider components.

Components As Data

In this section, we will see how components can add data to an entity. In the context of FXGL's Entity-Component model, a component is a part of an entity. A component can carry any data you wish for an entity to possess. We will appeal to the analogy of adding instance fields to the class definition in Java:

```
class Car {
    int moveSpeed;
    Color color;
}
```

So if we wanted to add a new property (field) possessed by a car, we would have to modify the preceding class. In games, objects commonly "lose" old properties and "gain" new properties, which means we need to have the ability to modify the preceding class at runtime, and this is not trivial.

Furthermore, let us consider deeper implications of using the `Car` class. In the previous example, the `Car` class contains an instance field `moveSpeed`, implying that it can move. Now, imagine that we also have a player character that, too, can move. We can add `moveSpeed` to the `Player` class. However, there are many other movable game objects. So perhaps we should use inheritance instead and have an abstract class `MovableGameObject`, or better still an interface `Movable`. That is good and all, but what if we want an object that sometimes can move and sometimes cannot? For example, a car controlled by the player can move, but it cannot move by itself. We cannot simply remove an interface and attach it back when we need it. It is clear that this problem does not have a trivial solution using inheritance.

The good news is that in computer science, there is typically more than one solution for any given problem. Inheritance is not the only tool in our arsenal. You may have heard the phrase "composition over inheritance." Well, components are somewhat an extreme version of that. They allow us to attach fields and methods at runtime – how cool is that? Consider the snippet of code in Listing 3-1, which is an equivalent of the `Car` class, but rewritten with components:

Listing 3-1. An example of adding, removing, and accessing a component

```
Entity entity = new Entity();
entity.addComponent(new MoveSpeedComponent());
entity.addComponent(new ColorComponent());
```

```
// some time later we can make it immovable
entity.removeComponent(MoveSpeedComponent.class);

// or change a component's value like a normal field
entity.getComponentOptional(ColorComponent.class).
ifPresent(color -> {
    color.setValue(Color.RED);
});
```

We can see that with components, the inheritance problem we described earlier becomes quite trivial to solve. This change in representation of a problem is a common technique in computer science. This technique allows to rephrase the problem in a way that is easy to solve by a known method. In addition, the approach that uses components isolates logic in each component. This means we can attach this `MoveSpeedComponent` to any entity. We can also add any number of different components to an entity, making this approach highly scalable. Finally, there is a strong adherence to the Single Responsibility principle (from SOLID principles), so all changes that need to modify, say, movement will take place inside a single component, namely, `MoveSpeedComponent`.

When creating an entity, it is preferred to add any components that you know an entity is going to possess up front. Then you can enable or disable components as necessary. For example, `CollidableComponent` signifies that an entity can collide with something. Suppose we do not want our entity to be collidable when created. Instead of adding the component when the entity becomes collidable, we can simply add the component at creation and set its value to false (in Java, we can use `var` instead of the type of the left side to shorten the code):

```
var c = new CollidableComponent();
c.setValue(false);
entity.addComponent(c);
```

Having considered the case where a component acts as a data container, we will now focus on how we can use components to add behavior to entities.

Components As Behavior

Components can also carry behavior information with them, effectively allowing entities to obtain new methods and be able to perform new actions. In fact, we previously demonstrated the analogy of adding a component with data as adding a field to a class. Well, adding components with behavior is like adding methods. Components allow us to make an entity do something, thus defining entity behavior. Before we create a custom component, let us first consider the built-in methods in the `Component` class:

- onAdded() – This callback is called when the component is added to an entity.
- onRemoved() – This callback is called when the component is removed from an entity.
- onUpdate() – This callback is called every frame to allow the component to update its logic.
- pause() – Calling this method will make the component pause processing any updates in onUpdate().
- resume() – Calling this method will make the component resume processing any updates in onUpdate().

Any custom class that extends from `Component` will have access to these methods.

Custom Components

Suppose we want entity to be a lift so that it can carry the player to the top of a mountain. As previously seen, adding such a component to an entity is trivial:

```
entity.addComponent(new LiftComponent());
```

Now we will consider how one might implement such a LiftComponent. Each component has an onUpdate() method that can be overridden to provide the necessary functionality (recall from Chapter 2 that such overrides are annotated with @Override). Using this approach, we can construct the piece of code shown in Listing 3-2 that will allow us to turn any entity into a lift.

Listing 3-2. An example of a custom component

```
public class LiftComponent extends Component {
    LocalTimer timer = ...;
    boolean isGoingUp = true;
    double speed = ...;

    @Override
    public void onUpdate(double tpf) {
        if (timer.elapsed(duration)) {
            isGoingUp = ! isGoingUp;
            timer.capture();
        }

        entity.translateY(isGoingUp? -speed * tpf :
        speed * tpf);
    }
}
```

You will note that we do not need to initialize the reference to the "entity" object. It is automatically injected into the component when we add it to an entity. This FXGL feature significantly simplifies the developer code. Another similar feature that avoids extra boilerplate code is component injection.

Component Injection

The common use case for automated component injection is when you have two components, namely, Component1 and Component2, such that Component2 needs to have a reference to Component1:

```
class Component1 extends Component {}
class Component2 extends Component {
    private Component1 comp;
}
```

In this case, you do not need to manually obtain a reference to Component1 in your Component2 class. The object will be automatically injected into the "comp" field by FXGL during component addition. It is important to note that Component2 requires Component1 to be present on the entity since it is a dependency. The check for required components is performed during component addition, so it is crucial to ensure that all required components are added first. In this case, Component1 should be added before Component2; otherwise, FXGL will raise an exception.

Let us see how such an injection works in practice. For instance, Listing 3-3 shows a very simple player component with methods up, down, left, and right that allow basic movement of the entity to which the component will be attached.

Listing 3-3. A possible implementation of the custom PlayerComponent class

```
public class PlayerComponent extends Component {

    // note that this component is injected automatically
    private TransformComponent position;

    private double speed = 0;
    @Override
    public void onUpdate(double tpf) {
        speed = tpf * 60;
    }

    public void up() {
        position.translateY(-5 * speed);
    }

    public void down() {
        position.translateY(5 * speed);
    }

    public void left() {
        position.translateX(-5 * speed);
    }

    public void right() {
        position.translateX(5 * speed);
    }
}
```

We can see that the `position` component field does not need to be initialized by the developer. It will be automatically injected by FXGL, thus reducing the amount of code.

Now that we are familiar with entities and components, it is time to focus on the game world, which is where entities spend most of their life cycle.

Game World

In this section, we will explore the game world and its significance in effective entity storage and query. The game world is, essentially, a data structure that contains entities. It is also responsible for adding, updating, and removing entities from the game. Finally, the game world provides means of querying (searching the world for) entities by various criteria. For example, you can collect all enemy entities, so you can boost their attributes when the player levels up.

Adding an entity to the world also adds it to the game. Similarly, to remove an entity from the game, all you need to do is to remove it from the world. Entities that are currently in the game world are considered to be active – their `isActive()` method will return true. Let us create a simple entity and add it to the game world. The game world in FXGL can be obtained by calling

```
var world = FXGL.getGameWorld();
```

To add an entity, `e`, to it, we can call

```
world.addEntity(e);
```

Finally, to remove the entity, we can call

```
world.removeEntity(e);
```

We are now able to add and remove entities as necessary, so our attention moves to game world queries.

World Queries

There are various ways to search for entities within the world. Typically, such methods begin with "get". For example, `getEntitiesByType()`. We will consider, in turn, querying entities by

- Type (requires TypeComponent) – Suppose we have an enum called EntityType that contains valid types for entities in your game. It might be that you wish to obtain all enemies in the game, so that you can apply a power-up to them, as a way to increase the game difficulty. This can easily be done by calling

  ```
  var enemies = world.
  getEntitiesByType(EntityType.ENEMY);
  ```

- ID (requires IDComponent) – Suppose you might allow copies of entities. This is especially useful for RPG-type games, where you have a lot of duplicates of items. So maybe we placed multiple forges in the game world and assigned a unique IDComponent to each forge. While the name is the same, "Forge", its numeric ID is different. The following code allows you to obtain a reference to a specific forge in the game:

  ```
  var forge123 = world.getEntityByID("Forge", 123);
  ```

- Singleton – There are many cases where there is just one entity of a given type, such as the player. It can be troublesome to go through a list of entities to get the one we want. So FXGL provides a nice utility query for that:

  ```
  var player = world.getSingleton(EntityType.PLAYER);
  ```

- Random – In gameplay, it is common to introduce some elements of randomness. If you wish to blow up a random enemy, you can get a random enemy entity via

  ```
  var enemy = world.getRandom(EntityType.ENEMY);
  ```

- Component – Some in-game logic should only be executed on entities that have certain components. An easy way to grab all entities with a given component is as follows:

  ```
  var entsWithComponent = world.getEntitiesByComponent(
  MyComponent.class);
  ```

- Range – Gameplay logic, such as triggering an interaction with an entity or dealing damage to entities in a radius, can be implemented by querying entities in a specific range. For example, to achieve this, we can call the following method to obtain all entities that fall inside a given rectangle (either fully or partially):

  ```
  var entitiesNearby = world.getEntitiesInRange(new
  Rectangle2D(5, 5, 30, 30));
  ```

- Filter – Finally, we can query entities in a general way by using a custom filter. The filter is essentially a function that given an entity returns true or false. If the result is true, then that entity will be added to the list to be returned:

  ```
  var entsWithHP10 = world.getEntitiesFiltered(e ->
  e.getInt("hp") == 10);
  ```

This completes the overview of the game world and its common functionality related to storing and querying entities. We will now focus on making use of these concepts in a Pong clone.

Pong

In this section, we will build a fully functioning foundation for a Pong game. The full up-to-date source code is available from `https://github.com/AlmasB/FXGLGames` under the Pong directory.

In Figure 3-1 (on the left image), you can see what we will achieve in this section. Admittedly, the game looks a bit bland, but you will learn a lot about the basics of the FXGL workflow and how to develop games in a predesigned templated way. Further into this book, you will learn about graphical effects, which will allow you to make this classic game a bit more visually appealing while staying true to the original aesthetics, as can be seen in Figure 3-1 (on the right image).

A common approach when developing FXGL games is to start by defining what types of entities we are planning to have in the game. In this example, we will have

- The player bat
- The enemy bat
- The ball
- Walls

As discussed before, we can define types as Java enums. An enum is like a special class that can only hold a predefined set of values. To implement these types, we can create the following enum:

```
public enum EntityType {
    PLAYER_BAT, ENEMY_BAT, BALL, WALL
}
```

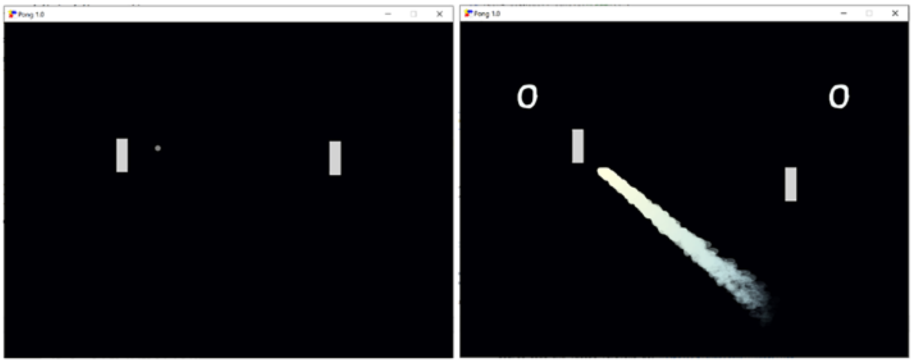


Figure 3-1. *A game of Pong: simple (left) and with visual enhancements (right)*

Building the Player Bat Component

As we saw earlier with components, the FXGL architecture promotes separation of concerns, which allows us to develop various parts of our game separately. This independent development of parts provides flexibility to developers and is in line with the Single Responsibility Principle (which is part of SOLID principles; more information can be found online). In light of this, we can start building the logic of our game entities by creating their components first. In Pong, there are two types of objects: bats (paddles) and the ball. We will begin with the `BatComponent`, the code for which is given in Listing 3-4.

Listing 3-4. BatComponent class

```
import com.almasb.fxgl.dsl.FXGL;
import com.almasb.fxgl.entity.component.Component;
import com.almasb.fxgl.physics.PhysicsComponent;

public class BatComponent extends Component {

    private static final double BAT_SPEED = 420;

    protected PhysicsComponent physics;
    public void up() {
        if (entity.getY() >= BAT_SPEED / 60)
            physics.setVelocityY(-BAT_SPEED);
        else
            stop();
    }

    public void down() {
        if (entity.getBottomY() <= FXGL.getAppHeight() - (BAT_
        SPEED / 60))
            physics.setVelocityY(BAT_SPEED);
        else
            stop();
    }

    public void stop() {
        physics.setLinearVelocity(0, 0);
    }
}
```

As with other code snippets, we will cover the `BatComponent` class in full. First, we note the three imports, which allow us to make use of FXGL methods, the generic `Component` class, and the `PhysicsComponent` class. To create a custom component, we define the name (here `BatComponent`) and extend from `Component`.

Next, we define our static constant value for the bat movement speed. It is declared as a variable at the top (rather than using it in the code as a hard-coded value) to make it easier to find the variable and modify the value later on should you wish to change it. Taking this line of thought further, you can even remove the “final” modifier if you would like to be able to change the variable at runtime.

Subsequently, we declare the `PhysicsComponent` reference, named physics. Recall from the "Component Injection" subsection in this chapter that we do not need to explicitly assign the reference to a value since during entity creation, the component will be automatically injected. Of course, we need to remember to add the `PhysicsComponent` before the `BatComponent` for this to work properly. We will come back to entity creation later on, and we will see how components are being added in the entity factory class.

The `BatComponent` has three methods: up, down, and stop. As their names imply, these methods are responsible for moving the bat entity. The implementation of these methods is trivial – we check if we can move in a direction (e.g., up) and do so if the check succeeds. More specifically, when attempting to move upward, we check if the Y value of the bat entity is further below than the top of the game (screen). This means there is space available to move. If the check fails, we stop the bat movement by setting its velocity to 0.

At this stage, you may be wondering about the `BAT_SPEED / 60` used in the check and not just `BAT_SPEED`. The reason is the difference between seconds and frames. The velocity we set using physics is `BAT_SPEED` per second, and there are typically 60 frames in a second. Therefore, each frame, we actually move `BAT_SPEED / 60` units. Hence, we use this value to check if there is sufficient space available for the bat entity to move.

Building the Enemy Bat Component

By extending (recall what inheritance is from Chapter 2) our `BatComponent`, we can define the `EnemyBatComponent` since most of the functionality remains the same. The `EnemyBatComponent` can be written as in Listing 3-5.

Listing 3-5. EnemyBatComponent class

```
import com.almasb.fxgl.entity.Entity;

public class EnemyBatComponent extends BatComponent {
    private Entity ball;

    @Override
    public void onUpdate(double tpf) {
        if (ball == null) {
            ball = entity.getWorld()
                           .getSingletonOptional(EntityType.BALL)
                           .orElse(null);
        } else {
            moveAI();
        }
    }

    private void moveAI() {
        Entity bat = entity;

        boolean isBallToLeft = ball.getRightX() <= bat.getX();

        if (ball.getY() < bat.getY()) {
            if (isBallToLeft)
                up();
            else
                down();
        } else if (ball.getY() > bat.getY()) {
            if (isBallToLeft)
                down();
            else
                up();
        } else {
            stop();
        }
    }
}
```

Let us examine this class in detail. You will note that this new component has a reference to the ball entity. At the start of the game, we don't know whether the ball is in the game world or not, so we initialize the ball inside the `onUpdate()` method. To do so, we check if the ball is `null` – at the start of the game, it is indeed `null`. So during the first frame the game runs, the ball entity type will be queried from the game world and initialized. By looking at the "else" branch of the if statement, we can see that a call to `moveAI()` is made. In essence, at a high level, this check can be described as follows: If the ball reference is null, initialize it; otherwise, move the enemy bat.

The AI of the enemy bat is actually quite simple. We will consider complex AI in a future chapter. In the meantime, our AI is just a few if statements. We know the enemy bat is on the right side of the screen. So we first check if the ball is to the left of the bat, meaning we should try and catch it. By contrast, if the ball is to the right, then we want to avoid touching it; otherwise, it will keep bouncing between the bat and the wall, constantly awarding points to the player bat. After this check, the only thing left to do is see if the ball is above or below the enemy bat and move accordingly.

Building the Ball Component

Having considered the logic of both bats, let us dive into the logic of the ball entity. The next component is the `BallComponent`, which is provided in Listing 3-6.

Listing 3-6. BallComponent class

```
import com.almasb.fxgl.entity.component.Component;
import com.almasb.fxgl.physics.PhysicsComponent;
import javafx.geometry.Point2D;

import static com.almasb.fxgl.dsl.FXGL.*;
import static java.lang.Math.*;

public class BallComponent extends Component {

    private PhysicsComponent physics;

    @Override
    public void onUpdate(double tpf) {
        limitVelocity();
        checkOffscreen();
    }

    private void limitVelocity() {
        // don't move too slow in X direction
        if (abs(physics.getVelocityX()) < 5 * 60) {
            physics.setVelocityX(signum(physics.getVelocityX())
            * 5 * 60);
        }

        // don't move too fast in Y direction
        if (abs(physics.getVelocityY()) > 5 * 60 * 2) {
            physics.setVelocityY(signum(physics.getVelocityY())
            * 5 * 60);
        }
    }
    // we use a physics engine, so it is possible to push
    the ball through a wall to outside of the screen, hence
    the check
    private void checkOffscreen() {
        var viewport = getGameScene().getViewport();
        var visArea = viewport.getVisibleArea();
        if (getEntity().getBoundingBoxComponent().
        isOutside(visArea)) {

            physics.overwritePosition(new Point2D(
                    getAppWidth() / 2,
                    getAppHeight() / 2
            ));
        }
    }
}
```

We again start by looking at the `onUpdate()` method. The high-level logic here is (a) limit the ball velocity so that it doesn't break the sound barrier and (b) check if the ball is off-screen. Each of these elements is implemented in their own methods. The `limitVelocity()` method checks the current velocity in the X axis and then in the Y axis, limiting the values to 5 * 60 as necessary. Finally, we check if the ball is outside of the screen. This can occur because we use the physics engine to run the collisions simulation for rigid bodies. It is possible, in some cases, that the ball gets pushed against the walls so hard that the correct value computation would put it outside of the wall. Hence, there is a simple check to test if the ball is within the visible area of the viewport. If it isn't, then just move the ball to the center of the screen.

Building the Entity Factory

Once we have the logic for each entity, implemented via components stated previously, we can now proceed with adding a factory to spawn the entities. The factory code is given in Listing 3-7, and it has two methods of note.

Listing 3-7. PongFactory class

```
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.entity.EntityFactory;
import com.almasb.fxgl.entity.SpawnData;
import com.almasb.fxgl.entity.Spawns;
import com.almasb.fxgl.entity.components.CollidableComponent;
import com.almasb.fxgl.physics.BoundingShape;
import com.almasb.fxgl.physics.HitBox;
import com.almasb.fxgl.physics.PhysicsComponent;
import com.almasb.fxgl.physics.box2d.dynamics.BodyType;
import com.almasb.fxgl.physics.box2d.dynamics.FixtureDef;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;

import static com.almasb.fxgl.dsl.FXGL.entityBuilder;
import static com.almasb.fxgl.dsl.FXGL.getip;

public class PongFactory implements EntityFactory {

    @Spawns("ball")
    public Entity newBall(SpawnData data) {
        PhysicsComponent physics = new PhysicsComponent();
        physics.setBodyType(BodyType.DYNAMIC);
        physics.setFixtureDef(new FixtureDef().density(0.3f).
        restitution(1.0f));
        physics.setOnPhysicsInitialized(() -> physics.
        setLinearVelocity(5 * 60, -5 * 60));
        return entityBuilder(data)
                .type(EntityType.BALL)
                .view(new Circle(5, 5, 5))
                .bbox(new HitBox(BoundingShape.circle(5)))
                .with(physics)
                .with(new CollidableComponent(true))
                .with(new BallComponent())
                .build();
    }

    @Spawns("bat")
    public Entity newBat(SpawnData data) {
        boolean isPlayer = data.get("isPlayer");

        PhysicsComponent physics = new PhysicsComponent();
        physics.setBodyType(BodyType.KINEMATIC);

        return entityBuilder(data)
                .type(isPlayer ? EntityType.PLAYER_BAT :
                EntityType.ENEMY_BAT)
                .viewWithBBox(new Rectangle(20, 60, Color.
                LIGHTGRAY))
                .with(new CollidableComponent(true))
                .with(physics)
                .with(isPlayer ? new BatComponent() : new
                EnemyBatComponent())
                .build();
    }
}
```

We can see the two methods in the factory class are `newBall()` and `newBat()`. We will look at these in order, starting from the `newBall()` method. Inside, we find a physics component since our ball is driven by the physics world. We need to initialize the physics to give the ball certain properties:

- The body type is set to dynamic (other types are static (no movement) and kinematic (player-controlled movement)), which allows the physics world to fully control the movement of the entity.
- Set the density and restitution (how bouncy it is) of the physics entity.
- On physics initialized, set linear velocity of the entity, so the ball starts moving.

Once our physics component is configured, we turn our attention to the entity builder. The builder allows us to quickly set up a bunch of properties of an entity. More specifically, we set the type to `EntityType.BALL`, the view to be a circle of radius 5, and the hit box to be of circular shape with radius 5. All of the components of an entity can easily be set by calling `with()`. Using this method, we add the physics, collidable, and ball components. This completes the construction of our ball entity, and we now move to constructing our bat entities.

Recall that we have two different types of the bat entity: player and enemy. When constructing the bat entity, we are able to pass an extra parameter to tell the factory whether it is the player bat or the enemy bat we wish to use. Hence, the first line of code inside the `newBat()` method extracts this parameter into a boolean `isPlayer`. As with the `newBall()` method, we also require a physics component. However, the configuration of the component is simpler – we just set the body type to be kinematic so that the player or AI can control the bats, rather than the physics world. All that is left is the construction of the bat entity using the entity builder. We set the entity type based on the boolean `isPlayer`. The view is a simple JavaFX rectangle, which is also used to automatically generate a hit box for the entity with the `viewWithBBox()` method. As with the ball entity, we set the collidable and physics components. Finally, using the boolean isPlayer, we can determine whether we should use BatComponent or EnemyBatComponent.

Building the Application Class

Finally, the class that brings all of these parts of the Pong game together is PongApp. Here, we will consider the class in detail, but as you progress through the book, in subsequent chapters, we will often omit the trivial details since they are already covered in these early chapters of the book.

As before, we will first take care of all the imports, given in Listing 3-8.

Listing 3-8. PongApp class imports

```
import com.almasb.fxgl.animation.Interpolators;
import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.core.math.FXGLMath;
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.entity.SpawnData;
import com.almasb.fxgl.input.UserAction;
import com.almasb.fxgl.physics.CollisionHandler;
import com.almasb.fxgl.physics.HitBox;
import com.almasb.fxgl.ui.UI;
import javafx.scene.input.KeyCode;
import javafx.scene.paint.Color;
import javafx.util.Duration;
import java.util.Map;
import static com.almasb.fxgl.dsl.FXGL.*;
```

Next, we will create the class PongApp, which extends GameApplication. This snippet is given in Listing 3-9. We will provide the basic settings such as the title and version of the game. Recall that the game width and height are automatically set to 800×600 by default; however, you can set your own values as necessary. We will also add the entry point to the game inside the main method.

Listing 3-9. PongApp class inherits from GameApplication

```
public class PongApp extends GameApplication {

    @Override
    protected void initSettings(GameSettings settings) {
        settings.setTitle("Pong");
        settings.setVersion("1.0");
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

We will need to access the player bat throughout the game, so we can control it. Therefore, it makes sense to create an instance-level variable to store the player bat component:

```
private BatComponent playerBat;
```

Once we have the reference to the player bat, we would like to be able to move it up and down. In other words, we need input, which we will handle in `initInput()`, given in Listing 3-10. You can see that instead of the simple input handling presented in the previous chapter, we use a slightly more sophisticated approach via the `getInput()` method. The advantage of utilizing this expanded API is that we can use `onActionBegin()`, `onAction()`, and `onActionEnd()` callbacks, which fire when a key is first pressed (action begin), held (action), and released (action end), respectively.

Listing 3-10. PongApp class input handling

```
@Override
protected void initInput() {
    getInput().addAction(new UserAction("Up") {
        @Override
        protected void onAction() {
            playerBat.up();
        }

        @Override
        protected void onActionEnd() {
            playerBat.stop();
        }
    }, KeyCode.W);

    getInput().addAction(new UserAction("Down") {
        @Override
        protected void onAction() {
            playerBat.down();
        }

        @Override
        protected void onActionEnd() {
            playerBat.stop();
        }
    }, KeyCode.S);
}
```

As we can see, our input is quite minimal – we only need the W and S keys to make the player bat move up or down. The `onAction()` callback is called while the key is being pressed, whereas the `onActionEnd()` is called when the key is released. These calls are exactly what we need in order to make the bat move or stop.

We also need to keep track of scores both for the player and the enemy (or player 2). We can do so by creating FXGL variables. Recall that they are almost like normal variables you would create in Java but have additional features, such as being able to listen to their changes and bind to them. Essentially, FXGL variables are like JavaFX properties. In Listing 3-11, we create two FXGL variables: `player1score` and `player2score`, both initialized to 0. Later, in `initGame()` (also in Listing 3-11), we listen to changes in these variables and declare the winner based on the values – a score of 11 is required to win the game.

Listing 3-11. Initializing the game and variables

```
 @Override
 protected void initGameVars(Map<String, Object> vars) {
     vars.put("player1score", 0);
     vars.put("player2score", 0);
 }
 @Override
 protected void initGame() {
getWorldProperties().<Integer>addListener("player1score",
(old, newScore) -> {
         if (newScore == 11) {
             showGameOver("Player 1");
         }
     });
     getWorldProperties().<Integer>addListener("player2sco
     re", (old, newScore) -> {
         if (newScore == 11) {
             showGameOver("Player 2");
         }
     });
```

```
        getGameWorld().addEntityFactory(new PongFactory());

        getGameScene().setBackgroundColor(Color.rgb(0, 0, 5));

        initScreenBounds();
        initGameObjects();
    }
```

Furthermore, we also add our PongFactory which we implemented earlier and set the game background color to RGB 0, 0, 5, which is very close to black. There are three methods in initGame() that we have not covered yet: showGameOver(), initScreenBounds(), and initGameObjects(). You should be able to guess what these methods do by considering their name. We will go through them in order in Listing 3-12.

Listing 3-12. Gameplay methods

```
    private void showGameOver(String winner) {
        getDialogService().showMessageBox(winner + " won! Demo
        over\nThanks for playing", getGameController()::exit);
    }

    private void initScreenBounds() {
        Entity walls =
                entityBuilder()
                .type(EntityType.WALL)
                .collidable()
                .buildScreenBounds(150);

        getGameWorld().addEntity(walls);
    }
    private void initGameObjects() {
        Entity ball = spawn("ball", getAppWidth() / 2 - 5,
        getAppHeight() / 2 - 5);
        Entity bat1 = spawn("bat", new SpawnData(getAppWidth()
        / 4, getAppHeight() / 2 - 30).put("isPlayer", true));
        Entity bat2 = spawn("bat", new SpawnData(3 *
        getAppWidth() / 4 - 20, getAppHeight() / 2 - 30).
        put("isPlayer", false));

        playerBat = bat1.getComponent(BatComponent.class);
    }
```

The implementation of the `showGameOver()` method is straightforward. We show a message box displaying who won the game, and once the box is dismissed, we exit the game. Next, in `initScreenBounds()`, we wish to create walls around the screen so that the ball can hit these walls and bounce back into the playable area of the game.

The last method in Listing 3-12 is `initGameObjects()`, where we simply create all of the entities that can be seen in the game: the ball and the two bats. We also remember to pass the `isPlayer` boolean variable when we create the bats. This is because our factory needs to know which one is the player bat and which one is the enemy bat. The last line in this method obtains a reference to the `BatComponent` to initialize the `playerBat` variable we defined earlier.

All that remains now is to set up the physics aspect of our game. Since all of our entities have `PhysicsComponent` attached, they are managed by the physics world (more on this will be presented in the following chapter). This means any setting we apply to the physics world will also affect these entities. For example, by default, the physics world applies a typical gravity vector, which has 0 velocity along the X axis and ~10 velocity along the Y axis. We do not need this in a game of Pong; otherwise, the ball would be heavily drawn toward the bottom of the screen. Let us consider the implementation of `initPhysics()` in Listing 3-13.

Listing 3-13. Initializing physics

```
    @Override
protected void initPhysics() {
    getPhysicsWorld().setGravity(0, 0);

    getPhysicsWorld().addCollisionHandler(new
    CollisionHandler(EntityType.BALL, EntityType.WALL) {
        @Override
        protected void onHitBoxTrigger(Entity a, Entity b,
        HitBox boxA, HitBox boxB) {
            if (boxB.getName().equals("LEFT")) {
                inc("player2score", +1);
            } else if (boxB.getName().equals("RIGHT")) {
                inc("player1score", +1);
            }
        }
    });
}
```

The very first thing we do in the method is effectively disable gravity in any direction. We then set up collision handlers – these are callback methods that are invoked when a collision between two entity types occurs. For example, in the preceding case, we wish to know when a collision occurs between a ball and the walls. Based on this, we can appropriately increase the score of player 1 or player 2. To add a collision handler, we first construct it by passing the two entity types (here `EntityType.BALL` and `EntityType.WALL`), and then we override the `onHitBoxTrigger()` method so that we can check which side of the wall the ball hit. As for incrementing the score variables, it is straightforward – we call the `inc()` method and provide the amount by which to increment the variable.

Game Structure Review

This completes the implementation of the Pong clone. You should now be able to run the application and see the entire functionality of the game we have implemented in this section. We will now briefly review the high-level architecture of the application we built to recap:

- There are three components: PlayerBatComponent, EnemyBatComponent, and BallComponent. Each component is attached to different entities to turn them (and their behavior) into the player bat, enemy bat, and ball, respectively.
- There is (always) one entity factory class named PongFactory. Its only responsibility is to create and configure entities in the game.
- There is (always) one application class named PongApp. It is responsible for bringing all other classes together and facilitating the program entry point (the main() method).

The `PongApp` class also initializes the game in the following order:

- initSettings() to set up the game settings, such as game dimensions, game version, and the title
- initInput() to set up input bindings for the user to be able to control the player entity
- initGameVars() to set up game variables that are accessible throughout the engine and game code
- initGame() to set up the PongFactory, game entities, and gameplay logic
- initPhysics() to set up the collision handlers and physics properties

Summary

In this chapter, we covered the theory behind the Entity-Component model and explored how this model can provide powerful abstractions of game state. In particular, entities allow us to represent game objects, whereas components provide data and behavior to entities. Furthermore, we explored various ways to query entities from the game world – a container for entities in the game. Finally, we used this theory in practice by implementing a simple yet insightful clone of the Pong game. In the next chapter, which is focused on game physics, we will consider collision-driven interaction between entities in more detail.

CHAPTER 4

Physics Case Study: Platformer Game

In this chapter, we will consider how important game physics is in the development of games. We will also see how physics is implemented in the FXGL game engine and how it can be used in FXGL games. More specifically, game physics typically serves two purposes: collision detection and simulations, both of which will be covered in the context of FXGL. By completing this chapter, you will learn the following:

- Commonly used collision detection techniques and how to implement them
- How to add collision handling code
- Physics world and its properties
- How to implement physics simulations
- How to build a simple platformer game

In many modern games, a lot of the interaction between the player and the game world is done via physics. For example, the player character will approach an in-game object and a prompt will appear as the physics subsystem will detect the proximity between these two entities. Another example is dropping a weighted item onto a pressure plate so that a physics property of the item (in this case, mass) will trigger the pressure

© Dr. Almas Baimagambetov 2022
A. Baimagambetov, *Learn JavaFX Game and App Development*,
https://doi.org/10.1007/978-1-4842-8625-8_4

plate. In this chapter, we will see how to implement similar examples, starting with collision detection. For simplicity, we will only focus on 2D.

Collision Detection

Collision detection tells us whether two entities are colliding (overlapping) or not colliding. As mentioned previously, this is a useful tool for enabling interaction between game objects. For example, the player entity collides with a coin entity and as a result of this collision, the player is able to pick up the coin. Figure 4-1 illustrates this scenario where the player is a blue square and the coin is a yellow circle.

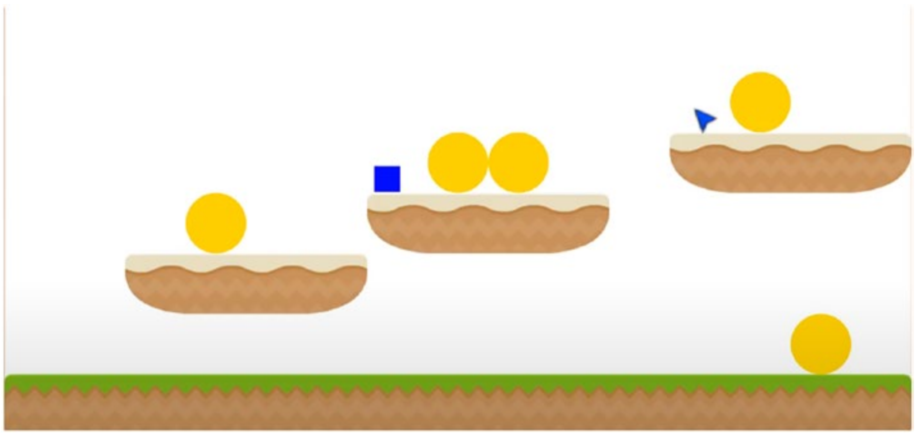


Figure 4-1. *An example platformer demonstrating possible entity interaction*

Collision detection techniques fall into two categories: broad phase and narrow phase. The goal of the broad-phase technique or algorithm is to avoid performing expensive computations for entities that are far away (and therefore cannot collide). These techniques are typically fast but less precise; that is, they are only good for quickly filtering out pairs of entities that cannot collide. Broad-phase techniques do not typically tell us if two entities are colliding; hence, once we have pairs that can potentially collide, we employ a narrow-phase technique. Compared to broad phase, these are slower but incredibly accurate, allowing us to identify whether a collision between two entities has occurred.

For broad phase, FXGL uses a grid-based spatial indexing algorithm to improve the performance of collision detection. Most of the broad-phase techniques, such as quadtree and k-d tree, utilize some form of spatial indexing to group entities that cannot collide based on the distance between them. However, for conciseness, we do not consider broad-phase algorithms in this book. Instead, we will focus on narrow-phase techniques.

A variety of collision detection techniques are available for computing whether two entities overlap in 2D, including

- Axis-aligned bounding box (each entity is wrapped with a rectangle that bounds the entity, and then we check if the rectangles overlap)
- Separating axis theorem (using several predefined axes, we draw lines of separation; if there exists at least one such line, there is no overlap)
- Circle-circle intersections (enclose each entity with a circle and check if the circles are overlapping)
- Polygonal (enclose each entity with a polygonal, typically convex, shape and check if these shapes overlap)
- Image pixel based (if entities have images as their views, then identify the intersection area of images, and if there is at least one pixel in the intersection, there is a collision between entities)

- Grid based (if the game is grid-based, then check if entities occupy the same cell or adjacent cells)

The preceding list should give you an idea with respect to the general approach: enclose entities with some known shape for which it is reasonably easy to detect collisions. In this book, we will only consider the first two techniques, which are commonly used methods in game development: axis-aligned bounding box and separating axis theorem. We will not go into too much detail since FXGL already implements these approaches and provides an easy way to access them.

The axis-aligned bounding box (AABB) algorithm is essential in any collision detection technique due to its performance and ease of implementation. The algorithm can be described as follows. Wrap each entity with a bounding box, which is the smallest rectangle that can completely enclose the entity. Next, check if two rectangles (boxes that wrap the two entities in question) overlap. If they do, register the collision; if they do not, then there is no collision.

A straightforward way to implement the AABB algorithm is as follows:

```
return box2.maxX >= box1.minX &&
       box2.maxY >= box1.minY &&
       box2.minX <= box1.maxX &&
       box2.minY <= box1.maxY
```

If all of these checks succeed, then there is a collision between `box1` and `box2`; thus, there is a collision between entities that these boxes represent.

However, there are limitations to the AABB approach. The main limitation is that bounding boxes, as the technique name implies, have to be axis-aligned. This means that if at least one entity is rotated, that is, has a nonzero rotation angle, then the approach is not suitable. In such cases, we can utilize a different technique called separating axis theorem, known as SAT for short. The SAT states that if there exists an axis along which there is a separation between two objects, then these objects are not overlapping. Unlike the bounding box approach given previously, this approach allows us to check for collisions even if entities are rotated (at nonzero degree angles). For example, in Figure 4-2, the collision between the two rotated entities on the right will be correctly reported by SAT.

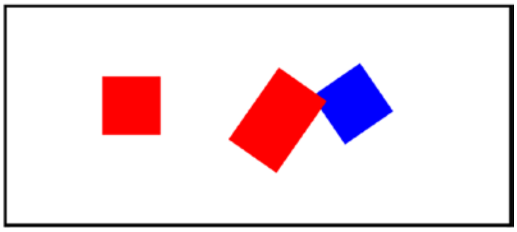


Figure 4-2. *Zero and nonzero rotation angles of entities*

In FXGL, both of these approaches are isolated behind a user-friendly API, like with most of the algorithms that are built-in. Suppose we have two entities, `e1` and `e2`. To check if they are colliding, in code, one can call the method `e1.isColliding(e2)`, which returns an intuitive boolean value.

Now that we have discussed the fundamental theory behind collision detection, it is time to turn our attention to handling these collisions.

Collision Handling

In this section, we examine the FXGL API that allows us to handle collisions, that is, to call gameplay code to perform some in-game operations as a result of a collision. We have used the player-coin example collision earlier, so we will continue building on that. As mentioned before, when the player collides with the coin, we would normally perform three actions: remove the coin, increase the player score, and play a sound effect to let the user know that an event has occurred. In the previous chapter, we already used some simple collision handling DSL API, but here, we will explore the full API, as it provides us with more options. To implement this gameplay logic, we create a class that extends a `CollisionHandler`. In the constructor, we specify the two types of entities for which this is a handler. We now consider the overridden methods.

Listing 4-1. An example of a collision handler

```
public class PlayerCoinHandler extends CollisionHandler {

    public PlayerCoinHandler() {
        super(EntityType.PLAYER, EntityType.COIN);
    }

    @Override
    protected void onHitBoxTrigger(Entity p, Entity c, HitBox
    boxA, HitBox boxB) { }

    @Override
    protected void onCollisionBegin(Entity p, Entity c) {
        c.removeFromWorld();
        FXGL.inc("score", +2);
        FXGL.play("pickup_coin.wav");
    }

    @Override
    protected void onCollision(Entity p, Entity c) { }

    @Override
    protected void onCollisionEnd(Entity p, Entity c) { }
}
```

As we can see from Listing 4-1, there are four such methods (if you are curious about all possible physics-related methods, they are available from `https://github.com/AlmasB/FXGL/tree/dev/fxgl-entity/src/main/java/com/almasb/fxgl`):

- onHitBoxTrigger() – This is the first method to be called when the collision occurs. It allows us to check which hit boxes collided. This is useful, for example, in fighting games: depending on where the character was hit, we can decide how many hit points to deduct.
- onCollisionBegin() – This is the method to be called in the same frame when the collision occurs, and it is called only once. Normally, this is the callback that is used for collectible items, such as a coin. In the example code, we implement just what we described the gameplay should be: remove the coin, increase the score, and play the sound.
- onCollision() – This is the method to be called every frame while the collision is present. This can be used to constantly deal damage to the player who is colliding with spikes. Alternatively, the callback can also be used to award points to the player for being in the scoring zone.
- onCollisionEnd() – This is the method to be called in the same frame when the collision stopped occurring. This can be used to make user interaction prompts invisible, as the player has left the area where the interaction is meaningful.

For the collision handler to work, two entities must satisfy the following requirements; they must both

- Have a CollidableComponent attached
- Have a bounding box
- Have a type

- Have the collision handler attached to the physics world by calling addCollisionHandler()

If these requirements are met, then the collision handler will appropriately handle the collisions (call the overridden methods) that occur between the two entities. Finally, when no longer needed, you can remove it by calling `removeCollisionHandler()`.

Physics World

So far, we have considered entities that are controlled by the developer. There is one other way to involve the physics world in gameplay, which is to let it control the entities. In this case, we are effectively running a simulation, such as rigid body dynamics. In order to run the simulation, an entity needs to have a `PhysicsComponent` attached. There are several commonly used configuration options that can be set via this component:

- setFixtureDef() – A fixture definition provides physics properties of an entity, such as friction, restitution (elasticity), density, and shape. Common shapes are rectangle, circle, chain, and polygon.
- setOnPhysicsInitialized() – Allows the developer to provide a callback, which is invoked when physics has been initialized for the entity.
- setLinearVelocity() – Allows the developer to manually set the velocity of a physics-supported entity. The velocity will then be applied by the physics world.
- setBodyType() – One of static, kinematic, or dynamic. A static entity has zero mass and zero velocity. These are typically objects that do not move, for example, platforms and walls. A kinematic entity has zero mass

but nonzero velocity that can be set by the developer. Kinematic entities are moved by the physics world and typically represent player-controlled objects, such as the bat in Pong or Breakout. A dynamic entity has a positive mass and nonzero velocity calculated by the forces being applied to the entity. Like kinematic, a dynamic entity is also moved by the physics world. They normally represent objects that do not need to be controlled by the player, for example, bullets or the ball in Pong or Breakout.

In rigid body dynamics, entities obtain a rigid (solid) body. When two entities of this form collide, the physics engine will attempt to separate them so that entities do not occupy the same space. Once an entity is considered part of the physics world, there are several commonly used world options you may wish to configure. These are accessible from the physics world object:

- setGravity() – Sets the gravity that is applied to dynamic entities in the world.
- raycast() – You can "shoot" a line from anywhere in the world to a specific point. The method will return the first entity hit by that line. This is useful if you wish to know if an entity is in the line of sight of another.
- toMeters() / toPixels() – Allows convenient conversion between pixels and meters. Note that the game works in pixel-based units, whereas the physics world operates with meters.
- toPoint() / toVector() – As before, this method converts from a point / vector from one context (game or physics world) to another (physics world or game).

Platformer

In this section, we will put all we have learned about physics in this chapter into practice and build a demo shown in Figure 4-3. We will do so by developing a simple platformer game. Unlike the game of Pong developed in the previous chapter, the platformer game will make use of external resources – assets, which also include a pre-built level in Tiled Editor. To simplify development, we will use publicly available free-for-use images and sprite sheets. Please ensure you download and place the assets in the appropriate directories (the following link provides details on the directory structure of the assets). Having done that, you will be able to follow along with the implementation given in this section. The used assets, as well as the links to the original sources and authors of these assets, are available from the FXGLGames GitHub repository: `https://github.com/AlmasB/FXGLGames/tree/master/Mario`.

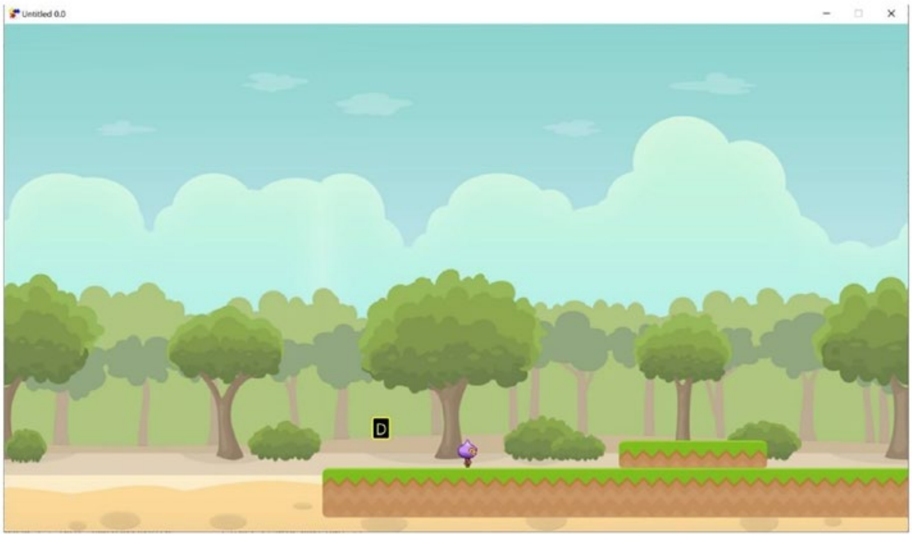


Figure 4-3. A platformer demo developed with FXGL

As per usual, the first step is to identify what entity types we will have in the game. In the platformer game we are developing, these are

```
public enum EntityType {
    PLAYER, COIN, PLATFORM, EXIT_TRIGGER, KEY_PROMPT, EXIT_
    SIGN, BUTTON, DOOR_TOP, DOOR_BOT
}
```

Some of these are self-explanatory, whereas others will make more sense as we go through the development.

We will now consider each of our classes in the game in the following order:

- PlayerComponent
- PlayerButtonHandler
- PlatformerFactory
- MainLoadingScene
- LevelEndScene
- PlatformerApp

Each of these classes manages one aspect of the game.

Game Logic

We proceed by examining the `PlayerComponent` class. Recall from the previous chapter that Components allow us to add data and behavior to entities. In addition, we can control entities via Components by adding methods. As before, we will go through the class in Listing 4-2 and its methods in order, providing the necessary details related to their purpose.

Listing 4-2. PlayerComponent class

```
import com.almasb.fxgl.entity.component.Component;
import com.almasb.fxgl.physics.PhysicsComponent;
import com.almasb.fxgl.texture.AnimatedTexture;
import com.almasb.fxgl.texture.AnimationChannel;
import javafx.geometry.Point2D;
import javafx.scene.image.Image;
import javafx.util.Duration;

import static com.almasb.fxgl.dsl.FXGL.image;

public class PlayerComponent extends Component {

    private PhysicsComponent physics;

    private AnimatedTexture texture;

    private AnimationChannel animIdle;
    private AnimationChannel animWalk;

    private int jumps = 2;

    public PlayerComponent() {
        Image image = image("player.png");

        animIdle = new AnimationChannel(image, 4, 32, 42,
        Duration.seconds(1), 1, 1);
        animWalk = new AnimationChannel(image, 4, 32, 42,
        Duration.seconds(0.66), 0, 3);

        texture = new AnimatedTexture(animIdle);
        texture.loop();
    }

    @Override
    public void onAdded() {
        entity.getTransformComponent().setScaleOrigin(new
        Point2D(16, 21));
        entity.getViewComponent().addChild(texture);

        physics.onGroundProperty().addListener((obs, old,
        isOnGround) -> {
            if (isOnGround) {
                jumps = 2;
            }
        });
    }

    @Override
    public void onUpdate(double tpf) {
        if (physics.isMovingX()) {
            if (texture.getAnimationChannel() != animWalk) {
                texture.loopAnimationChannel(animWalk);
            }
        } else {
            if (texture.getAnimationChannel() != animIdle) {
                texture.loopAnimationChannel(animIdle);
            }
        }
    }

    public void left() {
        getEntity().setScaleX(-1);
        physics.setVelocityX(-170);
    }

    public void right() {
        getEntity().setScaleX(1);
        physics.setVelocityX(170);
    }

    public void stop() {
        physics.setVelocityX(0);
    }

    public void jump() {
        if (jumps == 0)
            return;

        physics.setVelocityY(-300);

        jumps--;
    }
}
```

There are five fields, one constructor, and six methods in this class. The fields are the physics component (which we will add to the entity in the factory class later as per usual), the animated texture (which allows us to use a sprite sheet), two animation channels (one for idle animation and one for walk animation), and finally the number of jumps we allow the player to make. The constructor sets up the animations. Note that `image()` (the full call is `FXGL.image()`) can be used to load any image from `/assets/textures` as a JavaFX `Image`. Our image is called player.png; hence, this is the argument we pass into the `image()` method. Using the loaded image, we construct two animation channels and loop the idle animation.

In the `onAdded()` callback, which is called when this component (`PlayerComponent`) is added to the entity, we need to do three things: set the scale origin (so we can correctly turn the player left or right based on movement), add the animated texture, and register a listener that notifies when the player is on the ground, so we can reset the number of jumps. Note that we set the scale origin of the player entity to point 16,21, which is the center point of a single frame (32×42) of player animations.

In the next callback, `onUpdate()`, we identify which of the two animations (walk or idle) to loop. The physics component can tell us if the entity is moving along a particular axis. In our use case, moving along the X axis would mean the player is not idle. Therefore, if `isMovingX()` is true, we want to loop the walk animation, otherwise the idle animation.

As expected, the `left()` and `right()` methods have a similar implementation. Scaling the entity with a negative value in the X axis means the entity will face the other (left) direction. Doing the same with a positive value scales the entity to face right. The velocities along the X axis can be set based on the game feel. Setting the X velocity to 0 will stop the entity entirely, hence the name of the method `stop()`. The last method `jump()` checks if we can jump: if the number of jumps is 0, then we cannot. The actual jump implementation simply sets the Y velocity to a negative value. Note that the positive Y axis faces down in FXGL (and in JavaFX). We now have a functional player component.

Next, we will define our very first collision handler class in Listing 4-3. Note that we have already used anonymous collision handlers briefly and now we will properly isolate handling code into a separate class. The collision here will be handled between the player and the interactable button. When the player collides with the button, it will trigger a prompt to become visible, to indicate that the button can be pressed. When the collision ends, the key prompt will once again become invisible. We can turn visibility on and off by setting the opacity of the entity to 1 and 0, respectively. Note that it is also possible to call `setVisible(true / false)`, which will achieve the same visual effect.

Listing 4-3. PlayerButtonHandler class

```
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.physics.CollisionHandler;
import com.almasb.fxglgames.platformer.EntityType;

import static com.almasb.fxgl.dsl.FXGL.getGameWorld;
```

```
public class PlayerButtonHandler extends CollisionHandler {

    public PlayerButtonHandler() {
        super(EntityType.PLAYER, EntityType.BUTTON);
    }

    @Override
    protected void onCollisionBegin(Entity player,
    Entity btn) {
        Entity keyEntity = btn.getObject("keyEntity");

        if (!keyEntity.isActive()) {
            keyEntity.setProperty("activated", false);
            getGameWorld().addEntity(keyEntity);
        }

        keyEntity.setOpacity(1);
    }

    @Override
    protected void onCollisionEnd(Entity player, Entity btn) {
        Entity keyEntity = btn.getObject("keyEntity");
        if (!keyEntity.getBoolean("activated")) {
            keyEntity.setOpacity(0);
        }
    }
}
```

Compared to the game of Pong we implemented in the previous chapter, there are far more entity types in our platformer game. Hence, the factory class is quite a bit larger. Consider the factory implementation in Listing 4-4, which we will also examine method by method. While there is a lot to go through, you only need to focus on one method at a time.

Listing 4-4. PlatformerFactory class

```
import com.almasb.fxgl.dsl.components.LiftComponent;
import com.almasb.fxgl.dsl.views.ScrollingBackgroundView;
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.entity.EntityFactory;
import com.almasb.fxgl.entity.SpawnData;
import com.almasb.fxgl.entity.Spawns;
import com.almasb.fxgl.entity.components.CollidableComponent;
import com.almasb.fxgl.entity.components.IrremovableComponent;
import com.almasb.fxgl.input.view.KeyView;
import com.almasb.fxgl.physics.BoundingShape;
import com.almasb.fxgl.physics.HitBox;
import com.almasb.fxgl.physics.PhysicsComponent;
import com.almasb.fxgl.physics.box2d.dynamics.BodyType;
import com.almasb.fxgl.physics.box2d.dynamics.FixtureDef;
import com.almasb.fxgl.ui.FontType;
import javafx.geometry.Point2D;
import javafx.scene.CacheHint;
import javafx.scene.input.KeyCode;
import javafx.scene.paint.Color;
import javafx.util.Duration;

import static com.almasb.fxgl.dsl.FXGL.*;
import static com.almasb.fxglgames.platformer.EntityType.*;

public class PlatformerFactory implements EntityFactory {

    @Spawns("background")
    public Entity newBackground(SpawnData data) {
        return entityBuilder()
                .view(new ScrollingBackgroundView(texture("back
                ground/forest.png").getImage(), getAppWidth(),
                getAppHeight()))
                .zIndex(-1)
                .with(new IrremovableComponent())
                .build();
    }

    @Spawns("platform")
    public Entity newPlatform(SpawnData data) {
        return entityBuilder(data)
                .type(PLATFORM)
                .bbox(new HitBox(BoundingShape.
                box(data.<Integer>get("width"),
                data.<Integer>get("height"))))
                .with(new PhysicsComponent())
                .build();
    }

    @Spawns("exitTrigger")
    public Entity newExitTrigger(SpawnData data) {
        return entityBuilder(data)
                .type(EXIT_TRIGGER)
                .bbox(new HitBox(BoundingShape.
                box(data.<Integer>get("width"),
                data.<Integer>get("height"))))
                .with(new CollidableComponent(true))
                .build();
    }

    @Spawns("doorTop")
    public Entity newDoorTop(SpawnData data) {
        return entityBuilder(data)
                .type(DOOR_TOP)
                .opacity(0)
                .build();
    }
```

```
@Spawns("doorBot")
public Entity newDoorBot(SpawnData data) {
    return entityBuilder(data)
            .type(DOOR_BOT)
            .bbox(new HitBox(BoundingShape.
            box(data.<Integer>get("width"),
            data.<Integer>get("height"))))
            .opacity(0)
            .with(new CollidableComponent(false))
            .build();
}

@Spawns("player")
public Entity newPlayer(SpawnData data) {
    PhysicsComponent physics = new PhysicsComponent();
    physics.setBodyType(BodyType.DYNAMIC);
    physics.addGroundSensor(new HitBox("GROUND_SENSOR",
    new Point2D(16, 38), BoundingShape.box(6, 8)));

    // this avoids player sticking to walls
    physics.setFixtureDef(new FixtureDef().friction(0.0f));

    return entityBuilder(data)
            .type(PLAYER)
            .bbox(new HitBox(new Point2D(5,5),
            BoundingShape.circle(12)))
            .bbox(new HitBox(new Point2D(10,25),
            BoundingShape.box(10, 17)))
            .with(physics)
            .with(new CollidableComponent(true))
            .with(new IrremovableComponent())
            .with(new PlayerComponent())
            .build();
}
```

```
    @Spawns("exitSign")
    public Entity newExit(SpawnData data) {
        return entityBuilder(data)
                .type(EXIT_SIGN)
                .bbox(new HitBox(BoundingShape.
                box(data.<Integer>get("width"),
                data.<Integer>get("height"))))
                .with(new CollidableComponent(true))
                .build();
    }

    @Spawns("keyPrompt")
    public Entity newPrompt(SpawnData data) {
        return entityBuilder(data)
                .type(KEY_PROMPT)
                .bbox(new HitBox(BoundingShape.
                box(data.<Integer>get("width"),
                data.<Integer>get("height"))))
                .with(new CollidableComponent(true))
                .build();
    }

    @Spawns("keyCode")
    public Entity newKeyCode(SpawnData data) {
        String key = data.get("key");

        KeyCode keyCode = KeyCode.getKeyCode(key);

        var lift = new LiftComponent();
        lift.setGoingUp(true);
        lift.yAxisDistanceDuration(6, Duration.seconds(0.76));

        var view = new KeyView(keyCode, Color.YELLOW, 24);
        view.setCache(true);
        view.setCacheHint(CacheHint.SCALE);

        return entityBuilder(data)
                .view(view)
                .with(lift)
                .zIndex(100)
                .build();
    }

    @Spawns("button")
    public Entity newButton(SpawnData data) {
        var keyEntity = getGameWorld().create("keyCode",
        new SpawnData(data.getX(), data.getY() - 50).
        put("key", "E"));
        keyEntity.getViewComponent().setOpacity(0);

        return entityBuilder(data)
                .type(BUTTON)
                .viewWithBBox(texture("button.png", 20, 18))
                .with(new CollidableComponent(true))
                .with("keyEntity", keyEntity)
                .build();
    }
}
```

In FXGL, the entity factory class allows each entity creation to be conveniently isolated into its own method. We will now go through these methods in order, starting with the background entity. This entity forms an endless scrolling view based on an image that loops around. The built-in `ScrollingBackgroundView` class allows us to do just that. We also add a new `IrremovableComponent` that we have not used before. Since we have multiple levels in this platformer game, we will be destroying an existing level and setting a new one from time to time. Most levels have some shared entities, such as the background and the player. Hence, there is no need to recreate these entities every time we wish to change a level. For these cases, the `IrremovableComponent` is useful as it locks the entity (to which it is attached) from being removed from the world.

The next entity type is platform. A platform only has a bounding box and a physics component. By default, the physics component already has a static body, so there is no need for us to alter it. Note that we do not need to create a view for the platform since it is loaded directly from the Tiled ".tmx" level file.

The exit trigger is an entity that, when it collides with the player, enables (makes visible) the exit door to the next level. In games, triggers are typically not visible themselves and work simply as collision objects. As expected, we create a bounding box for it and add the collidable component.

With respect to the door entity, it is split into door top and door bottom since there are two tiles that together make up a door. You will note from the code that only the bottom part of the door is made collidable to avoid triggering collision with the player twice.

The player entity is one of the most complex ones in this game. First, we configure the physics component to have the dynamic body type, so that is affected by the physics forces. We also add a ground sensor, allowing the developer to know when the player has hit the ground, that is, the player entity is not in the air. To avoid the player sticking to walls, we also set the friction to 0. Setting it to a nonzero positive value is likely to affect the gameplay and feel for the game, so you can experiment with it to achieve suitable functionality. Another thing to note here is that the player has two bounding boxes: one for the head and the other for the body to represent the collision shapes more accurately. Finally, we attach collidable, irremovable, and player components, which is straightforward.

The exit sign entity adds a little bit of a dynamic interaction between the player and the game world. The idea is that when the player collides with the sign, it will light up. This is not an incredibly important feature in this use case; however, it should give you some ideas on what you can do in your own games.

The key prompt entity is there to help the player identify which key needs to be pressed to interact with the world. For example, recall any game you have played recently, whether 2D or 3D, where upon getting close to a door or an item, the game visually prompted you to press a specific key to interact. This is exactly what we are doing here with the key prompt entity.

The button entity is part of the gameplay, where the player needs to activate the button in order to progress through the level. In the levels we use, the button will allow the player to make the door visible, thus making progress to the next level possible. This completes the factory class overview, and we will now move on to other classes in our game example.

Game Visuals: Subscenes

To make our platformer example a little bit more game-like, we are going to introduce a few subscenes. Scenes and their interaction are beyond the scope of this book, so only a very brief introduction will be given here. There will be two custom scenes in the game: one to show during game loading and one to show during the end of a level.

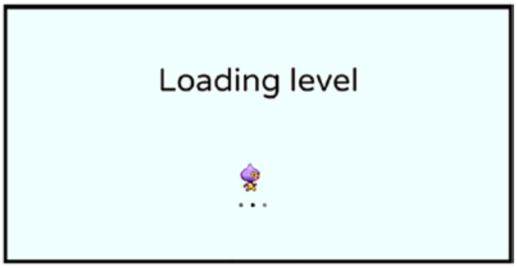


Figure 4-4. *A screenshot of the main loading scene*

As we can see in Figure 4-4, we create a JavaFX Text object with text “Loading level”, show the player image, and add three dots. This is implemented in Listing 4-5 as a single constructor call that sets up the animations used in the scene. The dots are animated using a built-in fade-in animation from FXGL (more on animations in Chapter 6). Finally, we position the text in the middle and add a player image. The image is animated to rotate from 0 to 360 degrees and loop indefinitely.

Listing 4-5. MainLoadingScene class

```
import com.almasb.fxgl.animation.Interpolators;
import com.almasb.fxgl.app.scene.LoadingScene;
import javafx.geometry.Rectangle2D;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.util.Duration;

import static com.almasb.fxgl.dsl.FXGL.*;

public class MainLoadingScene extends LoadingScene {

    public MainLoadingScene() {
        var bg = new Rectangle(getAppWidth(), getAppHeight(),
        Color.AZURE);

        var text = getUIFactoryService().newText("Loading
        level", Color.BLACK, 46.0);
        centerText(text,getAppWidth()/2,getAppHeight()/3 +25);

        var hbox = new HBox(5);

        for (int i = 0; i < 3; i++) {
            var textDot = getUIFactoryService().newText(".",
            Color.BLACK, 46.0);

            hbox.getChildren().add(textDot);

            animationBuilder(this)
```

```
                    .autoReverse(true)
                    .delay(Duration.seconds(i * 0.5))
                    .repeatInfinitely()
                    .fadeIn(textDot)
                    .buildAndPlay();
        }

        hbox.setTranslateX(getAppWidth() / 2 - 20);
        hbox.setTranslateY(getAppHeight() / 2);

        var t = texture("player.png").subTexture(new
        Rectangle2D(0, 0, 32, 42));
        t.setTranslateX(getAppWidth() / 2 - 32/2);
        t.setTranslateY(getAppHeight() / 2 - 42/2);

        animationBuilder(this)
                .duration(Duration.seconds(1.25))
                .repeatInfinitely()
                .autoReverse(true)
                .interpolator(Interpolators.EXPONENTIAL.EASE_
                IN_OUT())
                .rotate(t)
                .from(0)
                .to(360)
                .buildAndPlay();

        getContentRoot().getChildren().setAll(bg,text,hbox,t);
    }
}
```

The next scene we are going to introduce is the one that will be shown when the player completes any level. This scene will provide various bits of information to tell the player how well they did and how many stars they have achieved.

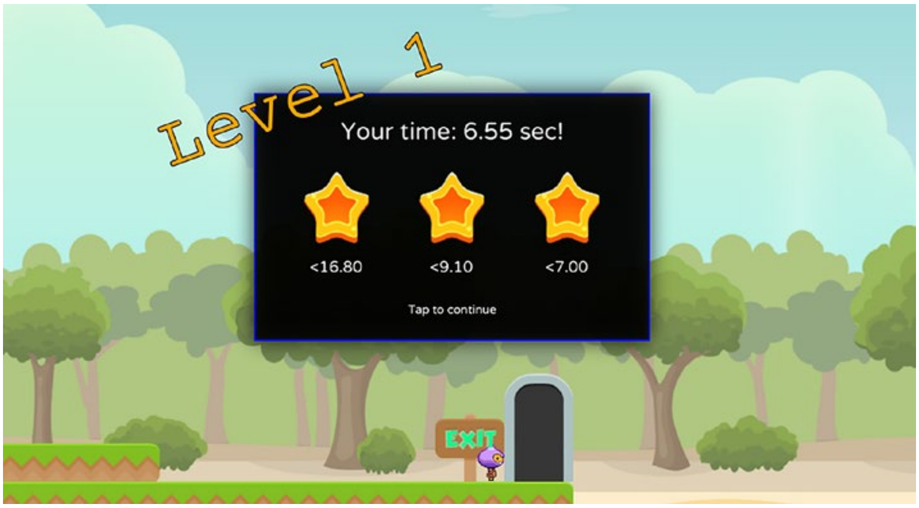


Figure 4-5. *A screenshot of the level end scene*

The majority of the code in `LevelEndScene`, which is given in Listing 4-6, sets up the view that is seen in Figure 4-5. As most of it stems from JavaFX, we will not go into much detail. The most important bit here is that the scene class extends a `SubScene`, which allows the scene to be added on top of an existing one, for example, on top of the game scene like in Figure 4-5, where both the game scene and the level end scene are visible. To add any visuals and UI, you can use normal JavaFX Nodes and add them to the scene content root, along with any animation you wish to apply.

Listing 4-6. LevelEndScene class

```
import com.almasb.fxgl.animation.Interpolators;
import com.almasb.fxgl.input.UserAction;
import com.almasb.fxgl.scene.SubScene;
import com.almasb.fxgl.texture.Texture;
import com.almasb.fxgl.ui.FontFactory;
import javafx.beans.property.BooleanProperty;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.geometry.Insets;
import javafx.geometry.Point2D;
import javafx.geometry.Pos;
import javafx.scene.effect.DropShadow;
import javafx.scene.input.MouseButton;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Priority;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.util.Duration;

import static com.almasb.fxgl.dsl.FXGL.*;

public class LevelEndScene extends SubScene {

    private static final int WIDTH = 400;
    private static final int HEIGHT = 250;

    private Text textUserTime = getUIFactoryService().
    newText("", Color.WHITE, 24.0);
    private HBox gradeBox = new HBox();

    private FontFactory levelFont = getAssetLoader().
    loadFont("level_font.ttf");
    private BooleanProperty isAnimationDone = new
    SimpleBooleanProperty(false);

    public LevelEndScene() {
        var bg = new Rectangle(WIDTH, HEIGHT, Color.color(0, 0,
        0, 0.85));
        bg.setStroke(Color.BLUE);
bg.setStrokeWidth(1.75);
bg.setEffect(new DropShadow(28, Color.
color(0,0,0, 0.9)));

VBox.setVgrow(gradeBox, Priority.ALWAYS);

var textContinue = getUIFactoryService().newText
("Tap to continue", Color.WHITE, 11.0);
textContinue.visibleProperty().bind(isAnimationDone);

animationBuilder(this)
        .repeatInfinitely()
        .autoReverse(true)
        .scale(textContinue)
        .from(new Point2D(1, 1))
        .to(new Point2D(1.25, 1.25))
        .buildAndPlay();

var vbox = new VBox(15, textUserTime, gradeBox,
textContinue);
vbox.setAlignment(Pos.CENTER);
vbox.setPadding(new Insets(25));

var root = new StackPane(
        bg, vbox
);

root.setTranslateX(1280 / 2 - WIDTH / 2);
root.setTranslateY(720 / 2 - HEIGHT / 2);

var textLevel = new Text();
textLevel.textProperty().bind(getip("level").
asString("Level %d"));
textLevel.setFont(levelFont.newFont(52));
textLevel.setRotate(-20);
```

```
        textLevel.setFill(Color.ORANGE);
        textLevel.setStroke(Color.BLACK);
        textLevel.setStrokeWidth(3.5);

        textLevel.setTranslateX(root.getTranslateX() -
        textLevel.getLayoutBounds().getWidth() / 3);
        textLevel.setTranslateY(root.getTranslateY() + 25);

        getContentRoot().getChildren().addAll(root, textLevel);

        getInput().addAction(new UserAction("Close Level End
        Screen") {
            @Override
            protected void onActionBegin() {
                if (!isAnimationDone.getValue())
                    return;

                getSceneService().popSubScene();
            }
        }, MouseButton.PRIMARY);
    }

    public void onLevelFinish() {
        isAnimationDone.setValue(false);

        Duration userTime = Duration.seconds(getd("levelTime"));

        LevelTimeData timeData = geto("levelTimeData");

        textUserTime.setText(String.format("Your time: %.2f
        sec!", userTime.toSeconds()));

        gradeBox.getChildren().setAll(
                new Grade(Duration.seconds(timeData.star1),
                userTime),
                new Grade(Duration.seconds(timeData.star2),
                userTime),
                new Grade(Duration.seconds(timeData.star3),
                userTime)
        );

        for (int i = 0; i < gradeBox.getChildren().
        size(); i++) {
            var builder = animationBuilder(this).
            delay(Duration.seconds(i * 0.75))
                    .duration(Duration.seconds(0.75))
                    .interpolator(Interpolators.ELASTIC.
                    EASE_OUT());

            // if last star animation
            if (i == gradeBox.getChildren().size() - 1) {
                builder = builder.onFinished(() ->
                isAnimationDone.setValue(true));
            }

            builder.translate(gradeBox.getChildren().get(i))
                    .from(new Point2D(0, -500))
                    .to(new Point2D(0, 0))
                    .buildAndPlay();
        }

        getSceneService().pushSubScene(this);
    }

    private static class Grade extends VBox {

        private static final Texture STAR_EMPTY =
        texture("star_empty.png", 65, 72).darker();
        private static final Texture STAR_FULL = texture("star_
        full.png", 65, 72);
        public Grade(Duration gradeTime, Duration userTime) {
            super(15);

            HBox.setHgrow(this, Priority.ALWAYS);

            setAlignment(Pos.CENTER);

            getChildren().add(userTime.lessThanOrEqualTo
            (gradeTime) ? STAR_FULL.copy() : STAR_EMPTY.
            copy());

            getChildren().add(getUIFactoryService().
            newText(String.format("<%.2f", gradeTime.
            toSeconds()), Color.WHITE, 16.0));
        }
    }

    public static class LevelTimeData {

        private final double star1;
        private final double star2;
        private final double star3;

        /**
         * @param star1 in seconds
         * @param star2 in seconds
         * @param star3 in seconds
         */
        public LevelTimeData(double star1, double star2, double
        star3) {
            this.star1 = star1;
            this.star2 = star2;
            this.star3 = star3;
        }
    }
}
```

You may consider the level-end-scene code given here just as an example, rather than reference implementation. Game visuals and aesthetics are not an exact science and require a lot of trial and error from the developer to get the game feel just right. Often, this involves feedback from human participants who can playtest your game and provide invaluable insights into game design.

Combining Logic and Visuals

We are now in a position where we can wire all of these classes, whether logic or visuals related, together with the `PlatformerApp` class. As per the standard architecture in FXGL, all of the functionality we developed will be used in a class called "App". In this case, it is `PlatformerApp`. It is a reasonably complex class, so we will tackle it in sections, covering methods one by one. The order is as follows:

- initSettings() – Initializes the game settings
- initInput() – Initializes user actions that control the game character
- initGameVars() – Initializes the game variables that are accessible globally
- initGame() – Initializes the rest of the game and gameplay
- initPhysics() – Initializes the physics settings and collision handlers
- makeExitDoor() – Makes the exit door visible in the game
- onUpdate() – Called every game frame to increment the game time and check for player's death

- nextLevel() – Advances the game to the next level
- setLevel() – Sets a given level by loading it from the Tiled ".tmx" file

All of these methods are provided as part of Listing 4-7. In `initInput()`, we have four keys we would like to bind to: A (to move left), D (to move right), W (to jump), and E (to use). We have previously used a more concise set of APIs for input; however, here, we need to use the full API so we can override `onAction()` and `onActionEnd()`. The actual input implementation is trivial for A, D, and W since we just call the player component's methods we created earlier. The implementation of E is a bit more complex. We would like to find all button entities in the game and then check that the button is collidable and is currently colliding with the player. If so, we update the button view, so it indicates that it has been pressed, and then we call `makeExitDoor()`, which we will cover shortly.

In `initGameVars()`, we initialize game variables to be accessible from anywhere in the source code. We store two things: the current level and the time spent on this level. Using the time value, at the end of the level, we can check and award stars appropriately in the level end scene, which we constructed earlier.

The next method is `initGame()`, where the majority of the game initialization process takes place. We add our platformer factory class, so FXGL knows how to create our entities. We create the level, spawn the player, and spawn the background (note that because of background z-index, we do not have to spawn the background before anything else; z-index nicely takes care of any ordering issues). Finally, we bind the viewport (visible area of the game) to the player entity. This last action makes the viewport follow the player entity and provide the side-scrolling effect you typically see in platformer games. We will discuss the viewport and its importance in Chapter 6.

In initPhysics(), we set up the physics configuration of the game. Specifically, we first set the gravity to an arbitrary value of 760 based on how the player controls feel. If the player appears too light, you may wish to increase the value. Similarly, if the player jumps are too short, you can decrease the value. Recall that when registering a collision handler, there are two approaches you can utilize: one is to create a collision handler class, which we did in PlayerButtonHandler above, or use DSL API provided by FXGL. When the player collides with the exit sign, the exit trigger, and the exit door, we only want to trigger the collision handling code once. Hence, we use onCollisionOneTimeOnly(). The code inside one-time collision handlers for these three entity types is straightforward. The last collision handler registered in initPhysics() is the one between the player and key prompts. We do not wish the collision between these types to happen only once, but we want to be notified every time the collision occurs; hence, we use onCollisionBegin(), which we saw before. The code inside the handler spawns a key prompt entity that floats for a few seconds indicating to the player that a specific key can be pressed to trigger some gameplay mechanic.

The makeExitDoor() method simply gets the references to the two entities (the top half and the bottom half) in the world that together form the exit door. We enable the collision registering on the door and make both parts visible in the game by setting their opacity to 1 (and therefore transparency to 0).

Inside onUpdate(), we increase the level time by a value equal to the time per frame, which is normally 0.016(6) if running at 60 frames per second. The only other code inside the method is a check for the player's Y value. If it is greater than the game height, then the player must have fallen through the level (by missing a platform jump), so we restart the same level.

The nextLevel() method is responsible for setting the next level or showing the game over message if there are no more levels. This check is performed first to see if our level is the maximum level we have in the game. Failing the check, we increment the level number by 1 and set it as the current level.

The setLevel() method is where the level generation happens. We reset the player's position to the beginning of the game and also reset the level time. The setLevelFromMap() method is provided by FXGL to automatically parse a ".tmx" file into an FXGL level. We can also query various properties associated with the ".tmx" file, such as the custom property of shortestTime needed to achieve three stars in the level. Finally, we construct a new level end scene based on this data.

To sum up Listing 4-7, the application class simply brings together all of the individual components, logic, and view-related classes that we implemented in other files. You will find this pattern (where functionality classes are developed first and then combined in the application class) throughout your development with FXGL.

Listing 4-7. PlatformerApp class

```
import com.almasb.fxgl.animation.Interpolators;
import com.almasb.fxgl.app.ApplicationMode;
import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.app.scene.GameView;
import com.almasb.fxgl.app.scene.LoadingScene;
import com.almasb.fxgl.app.scene.SceneFactory;
import com.almasb.fxgl.app.scene.Viewport;
import com.almasb.fxgl.core.util.LazyValue;
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.entity.SpawnData;
import com.almasb.fxgl.entity.components.CollidableComponent;
import com.almasb.fxgl.entity.level.Level;
import com.almasb.fxgl.input.UserAction;
import com.almasb.fxgl.input.view.KeyView;
import com.almasb.fxgl.input.virtual.VirtualButton;
import com.almasb.fxgl.physics.PhysicsComponent;
import javafx.geometry.Point2D;
import javafx.scene.input.KeyCode;
import javafx.scene.paint.Color;
import javafx.util.Duration;

import java.util.Map;

import static com.almasb.fxgl.dsl.FXGL.*;
import static com.almasb.fxglgames.platformer.EntityType.*;

public class PlatformerApp extends GameApplication {

    private static final int MAX_LEVEL = 5;
    private static final int STARTING_LEVEL = 0;

    private LazyValue<LevelEndScene> levelEndScene = new
    LazyValue<>(() -> new LevelEndScene());
    private Entity player;

    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(1280);
        settings.setHeight(720);
        settings.setSceneFactory(new SceneFactory() {
            @Override
            public LoadingScene newLoadingScene() {
                return new MainLoadingScene();
            }
        });
    }
```

```
@Override
protected void initInput() {
    getInput().addAction(new UserAction("Left") {
        @Override
        protected void onAction() {
            player.getComponent(PlayerComponent.
            class).left();
        }

        @Override
        protected void onActionEnd() {
            player.getComponent(PlayerComponent.
            class).stop();
        }
    }, KeyCode.A);

    getInput().addAction(new UserAction("Right") {
        @Override
        protected void onAction() {
            player.getComponent(PlayerComponent.class).
            right();
        }

        @Override
        protected void onActionEnd() {
            player.getComponent(PlayerComponent.class)
            .stop();
        }
    }, KeyCode.D);

    getInput().addAction(new UserAction("Jump") {
        @Override
        protected void onActionBegin() {
                player.getComponent(PlayerComponent.class)
                .jump();
            }
        }, KeyCode.W);

        getInput().addAction(new UserAction("Use") {
            @Override
            protected void onActionBegin() {
                getGameWorld().getEntitiesByType(BUTTON)
                        .stream()
                        .filter(btn -> btn.hasComponent(Col
                        lidableComponent.class) && player.
                        isColliding(btn))
                        .forEach(btn -> {
                            btn.removeComponent(CollidableCompo
                            nent.class);

                            Entity keyEntity = btn.getObject
                            ("keyEntity");
                            keyEntity.setProperty
                            ("activated", true);

                            KeyView view = (KeyView)
                            keyEntity.getViewComponent().
                            getChildren().get(0);
                            view.setKeyColor(Color.RED);

                            makeExitDoor();
                        });
            }
        }, KeyCode.E);
    }
```

```
@Override
protected void initGameVars(Map<String, Object> vars) {
    vars.put("level", STARTING_LEVEL);
    vars.put("levelTime", 0.0);
}

@Override
protected void initGame() {
    getGameWorld().addEntityFactory(new
    PlatformerFactory());

    player = null;
    nextLevel();

    // player must be spawned after call to nextLevel,
    otherwise player gets removed
    // before the update tick _actually_ adds the player to
    game world
    player = spawn("player", 50, 50);

    set("player", player);

    spawn("background");

    Viewport viewport = getGameScene().getViewport();
    viewport.setBounds(-1500, 0, 250 * 70, getAppHeight());
    viewport.bindToEntity(player, getAppWidth() / 2,
    getAppHeight() / 2);
    viewport.setLazy(true);
}

@Override
protected void initPhysics() {
    getPhysicsWorld().setGravity(0, 760);
    getPhysicsWorld().addCollisionHandler(new
    PlayerButtonHandler());
```

```
onCollisionOneTimeOnly(PLAYER, EXIT_SIGN, (player, 
sign) -> {
    var texture = texture("exit_sign.png").brighter();
    texture.setTranslateX(sign.getX() + 9);
    texture.setTranslateY(sign.getY() + 13);

    var gameView = new GameView(texture, 150);

    getGameScene().addGameView(gameView);

    runOnce(() -> getGameScene().
    removeGameView(gameView), Duration.seconds(1.6));
});

onCollisionOneTimeOnly(PLAYER, EXIT_TRIGGER, (player, 
trigger) -> {
    makeExitDoor();
});

onCollisionOneTimeOnly(PLAYER, DOOR_BOT, (player, 
door) -> {
    levelEndScene.get().onLevelFinish();

    // the above runs in its own scene, so fade will 
       wait until
    // the user exits that scene
    getGameScene().getViewport().fade(() -> {
        nextLevel();
    });
});

onCollisionBegin(PLAYER, KEY_PROMPT, (player, 
prompt) -> {
    String key = prompt.getString("key");
```

```
            var entity = getGameWorld().create("keyCode",
            new SpawnData(prompt.getX(), prompt.getY()).
            put("key", key));
            spawnWithScale(entity, Duration.seconds(1),
            Interpolators.ELASTIC.EASE_OUT());

            runOnce(() -> {
                despawnWithScale(entity, Duration.seconds(1),
                Interpolators.ELASTIC.EASE_IN());
            }, Duration.seconds(2.5));
        });
    }

    private void makeExitDoor() {
        var doorTop = getGameWorld().getSingleton(DOOR_TOP);
        var doorBot = getGameWorld().getSingleton(DOOR_BOT);

        doorBot.getComponent(CollidableComponent.class).
        setValue(true);

        doorTop.setOpacity(1);
        doorBot.setOpacity(1);
    }

    @Override
    protected void onUpdate(double tpf) {
        inc("levelTime", tpf);

        if (player.getY() > getAppHeight()) {
            setLevel(geti("level"));
        }
    }
    private void nextLevel() {
        if (geti("level") == MAX_LEVEL) {
            showMessage("You finished the demo!");
            return;
        }

        inc("level", +1);

        setLevel(geti("level"));
    }

    private void setLevel(int levelNum) {
        if (player != null) {
            player.getComponent(PhysicsComponent.class).
            overwritePosition(new Point2D(50, 50));
            player.setZIndex(Integer.MAX_VALUE);
        }

        set("levelTime", 0.0);

        Level level = setLevelFromMap("tmx/level" + levelNum  +
        ".tmx");

        var shortestTime = level.getProperties().
        getDouble("star1time");

        var levelTimeData = new LevelEndScene.LevelTimeData
        (shortestTime * 2.4, shortestTime*1.3, shortestTime);

        set("levelTimeData", levelTimeData);
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

This completes our platformer game example. We should be mindful that there are different ways to build any game. The case study covered in this chapter simply provides one possible implementation. As you learn more about JavaFX and FXGL, you will find different, possibly superior, ways to solve the same problem. This is common in game development and in the field of computer science in general. Nevertheless, based on this example, you should now be able to build a simple platformer game driven by physics.

Summary

In this chapter, we explored a range of physics uses in game development. We considered how to perform collision detection with a few different techniques. We also discussed their implementation details in FXGL. Next, we looked at some gameplay code that can be used for handling interactions between entities, for example, when a collision occurs. We saw how important collision handling is, since it allows the player to interact with the game world, as well as nonplayer entities to interact with one another. This importance is particularly prominent when considering 3D games, where interaction is fundamental to the immersion. Finally, we built a simple platformer game that utilized a lot of what we covered in this chapter. In the following chapter, we proceed by looking at how we can define AI for entities that are not player controlled.

CHAPTER 5

AI Case Study: Develop a Maze Action Game

In this chapter, we are going to look at the AI basics: what it helps us achieve in games and how to implement a range of AI behaviors in FXGL. AI is one of the key areas of game development. Without it, most single-player games would be far less engaging and fun. AI is what gives NPCs and enemies quasi-informed decision-making abilities with the goal to entertain the player. More specifically, we, as game developers, should aim to make the AI sophisticated enough that it can match the player's skill but shallow enough to give the player an invisible advantage. If designed and implemented well, such an AI agent will generate interesting and challenging interactions with the player but will know when to lose.

We will categorize AI into two: pathfinding and behavior. Pathfinding AI deals with searching a path between two points in a 2D (or 3D) plane. For example, a "seeker" enemy is able to hunt down the player while avoiding all obstacles inside the game level. Behavior AI is primarily focused on what an entity does at any given moment. In this chapter, we will explore several ways to implement behavior. By completing this chapter, you will learn about

© Dr. Almas Baimagambetov 2022

- Graph theory basics and the A* pathfinding algorithm
- How to use A* Component to find a path between two points in the game world
- How to use custom Components to define simple AI behavior
- How to use a finite state machine to define AI behavior with multiple states
- How to implement an action game using simple AI behavior

Pathfinding is typically done on a data structure that allows common computer science search algorithms to be applied. We begin this chapter by exploring one such data structure called a graph.

Graph Theory and Pathfinding Concepts

There exist a range of pathfinding algorithms based on graph searches. A graph is a collection of nodes and a collection of edges, which connect the nodes. An example can be seen in Figure 5-1, where there are ten nodes (circles) and also ten edges (lines). Graph theory is a fundamental area of computer science that studies graphs and its applications. There are many graph search algorithms tailored for specific tasks. In games, a common algorithm used for finding a path between two points is known as the A* (pronounced "A star") algorithm.

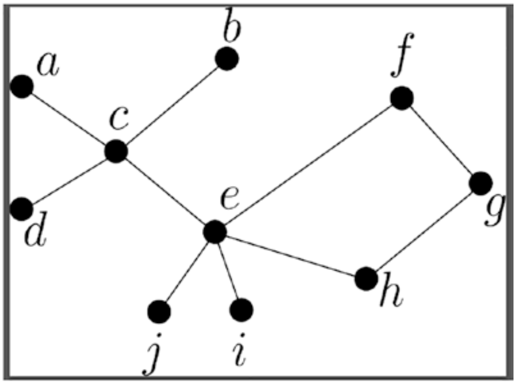


Figure 5-1. *A graph*

Before we can apply any search algorithm, including A*, the first task is to convert the game world into a graph. Suppose in your 2D game world you have rooms or cells that are interconnected, such that you can go from one room to its adjacent (neighboring) room. There will be cases where you cannot do this, for example, if a room is closed. A simple algorithm to convert such a world into a graph is to create a node for every room and an edge between two nodes, A and B, if you can go from A to B in a single unit of movement. The resulting structure is your graph. For example, if the graph in Figure 5-1 was produced in this way, then we can go from room "a" to room "c" in a single unit of movement. However, in order to go from "a" to "b", we need to get to "c" first.

Suppose in Figure 5-1 we wish to go from node "a" to node "g". Of course, we can use a brute force approach and simply check all possible routes from "a" to "g" and then pick a suitable one. On small graphs like this, the approach would generally be fine. However, imagine a vast world, like in GTA V, where the underlying graph would simply be too big to brute force, particularly given we have a fraction of a second to compute the path. It is clear that we need a more effective approach to solve such problems for the general case. This is where the A* pathfinding algorithm can be of tremendous help.

A* Pathfinding in FXGL

In order to use the A* algorithm in FXGL, we will first need to construct an AStarGrid object from our game world. AStarGrid is a 2D grid that consists of AStarCell objects, which can have one of the two states: WALKABLE or NOT_WALKABLE. Next, we need to add a few components to the entity we would like to move in the grid. The components are CellMoveComponent and AStarMoveComponent.

CellMoveComponent allows the entity to move in a cell-based environment, such as a grid, a graph, or a tile map. AStarMoveComponent makes use of CellMoveComponent and provides the necessary path information so that the entity does not go through walls or other obstacles. To help reinforce these theoretical concepts, we will build a small standalone application, whose code is in Listing 5-1. Before proceeding, do make a note of the imports required.

Listing 5-1. PathfindingSample class

```
import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.dsl.FXGL;
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.pathfinding.CellMoveComponent;
import com.almasb.fxgl.pathfinding.CellState;
import com.almasb.fxgl.pathfinding.astar.AStarGrid;
import com.almasb.fxgl.pathfinding.astar.AStarMoveComponent;
import javafx.scene.input.MouseButton;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.shape.StrokeType;

import static com.almasb.fxgl.dsl.FXGL.*;
```

```
public class PathfindingSample extends GameApplication {
    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(1280);
        settings.setHeight(720);
    }

    @Override
    protected void initGame() {
        var grid = new AStarGrid(1280 / 40, 720 / 40);

        var agent = entityBuilder()
         .viewWithBBox(new Rectangle(40, 40, Color.BLUE))
                .with(new CellMoveComponent(40, 40, 150))
                .with(new AStarMoveComponent(grid))
                .zIndex(1)
                .anchorFromCenter()
                .buildAndAttach();

        for (int y = 0; y < 720 / 40; y++) {
            for (int x = 0; x < 1280 / 40; x++) {
                final var finalX = x;
                final var finalY = y;

                var view = new Rectangle(40, 40, Color.WHITE);
                view.setStroke(Color.BLACK);
                view.setStrokeType(StrokeType.INSIDE);

                var e = entityBuilder()
                        .at(x * 40, y * 40)
                        .view(view)
                        .buildAndAttach();

         e.getViewComponent().addOnClickHandler(event -> {
```

```
if (event.getButton() == MouseButton.PRIMARY) {
                   agent.getComponent(AStarMoveComponent.class)
                   .moveToCell(finalX, finalY);

} else if (event.getButton() == MouseButton.SECONDARY) {
grid.get(finalX, finalY).setState(CellState.NOT_WALKABLE);
view.setFill(Color.RED);
}
                }); 
            }
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Let us examine the code in Listing 5-1 in detail. First, we can see that the AStarGrid is constructed with appropriate dimensions. Note that the values provided are in number of cells, rather than pixels, which is why we divide 1280 and 720 by 40 (size of a single cell in our case). Next, using EntityBuilder, we build our agent entity. You are already familiar with the process of building an entity from previous chapters, so we can skip the details. In this case, the entity is a blue square with CellMoveComponent and AStarMoveComponent. Note that the latter component needs to have access to the grid we constructed in order to be able to move intelligently around the level. As you can see, the actual grid traversal implementation details are hidden from the developer to make the API easy to use.

We will now build our entities that allow us, the users, to see into the game world and identify cells that are walkable and those that are not. Each entity will have a square view, whose color will be white by default. We will add an on-click handler to each entity's ViewComponent so that when we click on it, we can do two things: (1) if left click, we move our agent to that cell using its `AStarMoveComponent`; (2) if right click, we set the cell as one that cannot be walked on and also change its color to red. The left and right click events are standard JavaFX events, so you can use a similar setup for any of your game examples for quick prototyping.

Our small example is now complete. You should be able to run it and see a 2D grid with a blue square agent. Left-clicking on any cell will move the agent to that cell using the built-in A* pathfinding. Play around with it to see how right-clicking can create obstacles and how the agent will avoid them. An example screenshot is given in Figure 5-2, where the red obstacles that spell out "FXGL" block the blue agent's path.

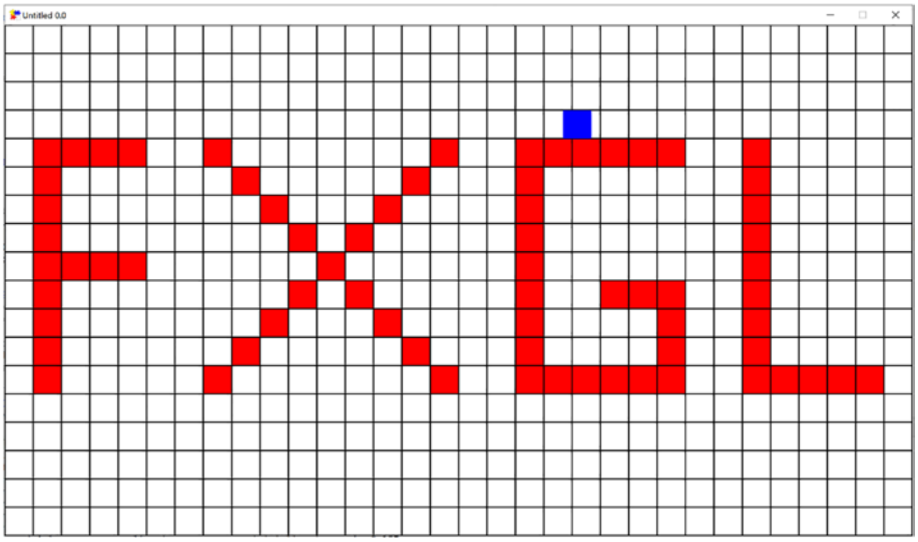


Figure 5-2. *An example application showing A* pathfinding in FXGL*

Behavior

We have covered how AI can move in a 2D world to reach its target using the world data. We now turn our attention to what AI should be doing when it does reach the target. In this section, we explore multiple ways of adding AI behavior to your enemy entities. While all are valid ways of adding AI, certain approaches are more suitable based on your use case. The most straightforward approach is to use a custom component when your entity behavior primarily has a single job. For example, suppose you have an enemy entity whose role is to be stationary and signal the other AI entities that the player is close. We can implement this behavior by creating a `SignalComponent`, shown in Listing 5-2.

Essentially, the component's `onUpdate()` method performs all of the logic behind this simple behavior. First, we will need to obtain the reference to the player entity so that we can check the distance between the player and us (the signaling entity). To do that, you can use one of the many query methods we covered in Chapter 3. Recall that `getEntity()`, when called from inside the component, returns the entity to which the component is attached, so the signaling entity. Using a simple if statement, we can check whether we should "signal other AI because the player is close." You can also add the else branch, which is not needed here, but it should give you an idea regarding how to implement similar use cases.

Listing 5-2. SignalComponent class

```
public class SignalComponent extends Component {
    @Override
    public void onUpdate(double tpf) {
        Entity player = ...
        if (getEntity().distance(player) < 100) {
            // constantly signal other AI that player is close
        }
    }
}
```

Let us consider another example in Listing 5-3. This AI behavior is focused on following any entity. Given a reference to any entity, such as the player, the onUpdate() method will constantly move the AI agent toward the direction of the entity. In general, you may have noticed that in FXGL, the update method is an appropriate place to handle entity logic and AI code.

Listing 5-3. FollowComponent class

```
public class FollowComponent extends Component {
    @Override
    public void onUpdate(double tpf) {
        Entity player = ...
        getEntity().translateTowards(player.getPosition(),10);
    }
}
```

As you can see from Listings 5-2 and 5-3, the behavior implementation in FXGL is quite straightforward. However, in games, there are various use cases that arise, for which this simple implementation is not as suitable. So far, we have covered behavior that primarily had a single job: to signal or to follow. When the AI behavior has two or more distinct states, then we can use an existing StateComponent to help us define the AI functionality in a more effective way. The StateComponent essentially allows us to use a finite state machine (FSM), where an AI agent can only be in one of predefined states. In order to do this in FXGL, first, add StateComponent to your entity. Next, implement your states, such as in Listing 5-4.

Listing 5-4. Defining an EntityState

```
private EntityState GUARD = new EntityState() {
    @Override
    protected void onUpdate(double tpf) {
        // code to run when in this state
    }
};
```

Finally, we can use StateComponent of the entity and set the created state: entity.getComponent(StateComponent.class).changeState(GUARD). From this point onward, the entity will be calling the onUpdate() from the GUARD state, allowing the developer to isolate behavior that is specific to that state, again, following the single responsibility principle. Any other states can be created in exactly the same way. You can see that this approach can promote flexibility since states do not necessarily rely on knowing the existence of other states. However, if you wish, like in the next example, you can still declare all states within a single file and switch to a new state from within an existing different state.

We will now look at a small FXGL application using FSM. In particular, let us imagine a resource gathering AI in a real-time strategy game, such as Warcraft 3, Stronghold Crusader, or many other games of the genre. Such an AI automatically moves to any assigned resource to gather it. Once its "backpack" is full, the AI will move to the nearest "stockpile" to deposit the gathered resources, thus clearing its backpack. Next, it will restart its loop and continue gathering resources. Such an AI component is useful in many real-time strategy games where "worker" type entities can gather resources for the player. Ultimately, we can identify three states: gathering, moving, and depositing. The first two states are continuous, meaning the entity will spend some time in them. However, depositing is a single action and does not necessarily need to be implemented as a state. The implementation of this AI can be seen in Listing 5-5. We now examine it in detail.

Listing 5-5. An example implementation of GathererComponent class

```
import com.almasb.fxgl.dsl.FXGL;
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.entity.component.Component;
import com.almasb.fxgl.entity.component.Required;
import com.almasb.fxgl.entity.state.EntityState;
import com.almasb.fxgl.entity.state.StateComponent;
import com.almasb.fxgl.time.LocalTimer;
import javafx.geometry.Point2D;
import javafx.util.Duration;

public class GathererComponent extends Component {

    private StateComponent stateComponent;
    private Entity target;
    private Point2D targetPosition;
    private Entity prevTarget;

    private EntityState GATHERING = new EntityState() {
        private LocalTimer gatherTimer = FXGL.newLocalTimer();
        private ResourceComponent resourceComponent;

        @Override
        public void onEnteredFrom(EntityState entityState) {
            resourceComponent = target.getComponent
            (ResourceComponent.class);
            gatherTimer.capture();
        }

        @Override
        public void onUpdate(double tpf) {
            if (isBackpackFull()) {
                FXGL.getGameWorld()
                        .getClosestEntity(entity, e ->
                        e.isType(EntityType.STOCKPILE))
                        .ifPresent(s -> sendTo(s));
                return;
            }
            if (resourceComponent.isEmpty()) {
                FXGL.getGameWorld()
                        // and matches resource type ...
                        .getClosestEntity(entity, e -> e.hasCom
                        ponent(ResourceComponent.class) && !e.
                        getComponent(ResourceComponent.class).
                        isEmpty())
                       .ifPresent(resource -> sendTo(resource));
                return;
            }

            if (gatherTimer.elapsed(Duration.seconds(2))) {
                resourceComponent.gather();
                entity.getProperties().increment
                (resourceComponent.getType().toString(), +1);

                gatherTimer.capture();
            }
        }
    };

    private EntityState MOVING = new EntityState() {
        @Override
        protected void onUpdate(double tpf) {
            // reached destination, so stop moving
            if (entity.getPosition().distance(targetPosition)
            < 5) {

                if (target != null) {


                    if (target.hasComponent(ResourceComponent.
                    class)) {
                        // target is a resource
                    stateComponent.changeState(GATHERING);

                } else if (target.isType(EntityType.
                STOCKPILE)) {
                    // target is a stockpile
                    depositResources(target);

                    if (prevTarget != null) {
                        sendTo(prevTarget);
                    } else {
                        // no target, just be idle
                        stateComponent.changeStateToIdle();
                    }
                }

            } else {
                // no target, just be idle
                stateComponent.changeStateToIdle();
            }

            return;
        }

        entity.translateTowards(targetPosition, 5);
    }
};

private boolean isBackpackFull() {
    return entity.getInt(ResourceType.WOOD.toString()) +
    entity.getInt(ResourceType.STONE.toString()) == 10;
}

private void depositResources(Entity stockpile) {
    for (var resourceType : ResourceType.values()) {
        int qty = entity.getInt(resourceType.toString());
```

```
            stockpile.getProperties().increment(resourceType.
            toString(), qty);

            entity.setProperty(resourceType.toString(), 0);
        }
    }

    public void sendTo(Entity target) {
        this.prevTarget = this.target;
        this.target = target;
        sendTo(target.getPosition());
    }

    public void sendTo(Point2D point) {
        targetPosition = point;

        stateComponent.changeState(MOVING);
    }
}
```

There are four variable fields and two state fields in this component:

- stateComponent – Reference to StateComponent, which is automatically injected.
- target – Current target of the entity, or null if there is no target.
- targetPosition – The position of the current target, or the point that the entity is moving toward if no target is present.
- prevTarget – The previous target of the entity.
- GATHERING state – In this state, the entity collects resources.

- MOVING state – In this state, the entity moves, either to a resource or to a stockpile.

In the `GATHERING` state, the `onUpdate()` overridden method first checks if the backpack is full. We need to perform this check since there is no point in gathering resources if we do not have space for them. If the check returns true, meaning the backpack is full, then we send the AI back to the "base." Next, we want to check if the resource is actually available to collect as it could have been depleted. If it is not available, we send our AI to the nearest matching resource entity to continue its gathering work. Having completed these checks, we check if the gathering timer has passed, since we do not want to gather the resource every frame, but every 2 seconds (~120 frames) in our example.

In the `MOVING` state, the `onUpdate()` method takes care of changing the position of our AI entity in the game world. We first check if the entity has reached its destination; if it has, then we check why we are here (to gather a resource, or to drop off the gathered resource); otherwise, we make the entity continue moving in the direction of its target.

The other methods in Listing 5-5 are there to simplify the implementation of our states, and you are encouraged to read through them on your own. So as you can see, at the end of the day, our AI is indeed just a sequence of conditional statements. However, if these are implemented properly and aligned with the appropriate concepts, such as FSM, then they create a powerful illusion for the player to play against. Now that we have covered different types of AI, as well as their simple implementations, we will proceed with a more specific game example.

Maze Action Game

In this section, we will build a simple action game with maze elements. Essentially, the game is a slightly modified clone of Pac-Man, which is a perfect example for this chapter. The game has four enemies in a maze-like level filled with coins. The player must traverse the maze and collect the coins while avoiding the enemies. Instead of making a complete clone, we will add some original elements to the game, such as the time bar and the teleport ability. All of the assets (including licenses under which they are being used) and code are available to download from the following link: `https://github.com/AlmasB/FXGLGames/tree/master/Pacman`. You can see an image from the game in Figure 5-3.

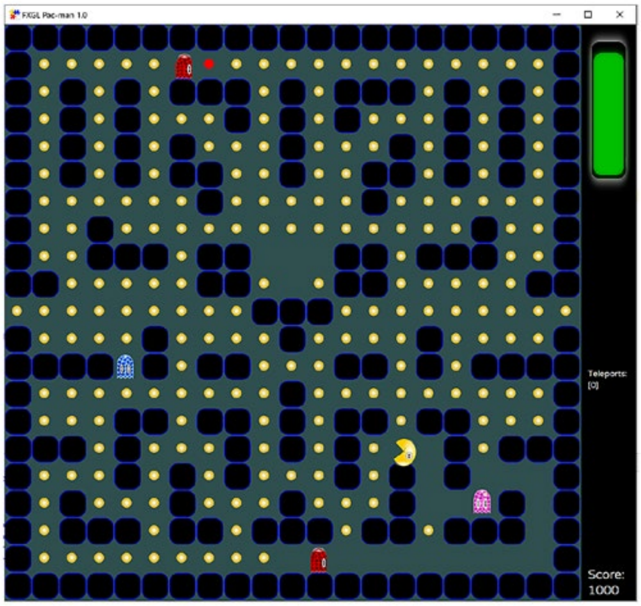


Figure 5-3. *A modified clone of Pac-Man*

Custom Components

As per usual, we start by defining entity types that are present in the game. In our Pac-Man clone, we only have four types:

```
public enum PacmanType {
    PLAYER, ENEMY, BLOCK, COIN
}
```

As this game is focused on AI, most of the functionality is contained within components. We mentioned earlier that there are four types of enemies. These are `GuardCoinComponent`, `DelayedChasePlayerComponent` (one with a delay and one without), and `RandomAStarMoveComponent`, which is available as a pre-built component from FXGL. We will consider the implementation of the first two files. As for the `RandomAStarMoveComponent`, it has only one simple goal, which is to randomly move the entity inside the level using the A* pathfinding. Therefore, we will not be discussing its implementation details here. However, you are welcome to browse through the component's source code inside the FXGL repository on GitHub.

With respect to custom component classes, there are four of them in this game example. They are

- GuardCoinComponent – This is an AI component to make an enemy entity to guard a chosen coin in the game.
- DelayedChasePlayerComponent – This is an AI component to make an enemy chase the player entity. This component has two modes: immediate (without a delay) and delayed. In the immediate mode, it will directly go to the player using the quickest route, whereas in the delayed mode, it will pick a slightly longer route, giving the player some breathing space.
- PaletteChangingComponent – This is not an AI component per se, but it is there to help the AI fool the player by changing the color scheme of each enemy so that the player cannot readily know which AI they are facing.

- PlayerComponent – This component allows the player entity to perform gameplay actions, such as move and teleport.

We will then go through the four components in the order mentioned previously, with the first one being the `GuardCoinComponent`, in Listing 5-6.

Listing 5-6. The implementation of the Guard AI in our game example

```
import com.almasb.fxgl.core.collection.grid.Cell;
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.entity.component.Component;
import com.almasb.fxgl.entity.component.Required;
import com.almasb.fxgl.pathfinding.astar.AStarMoveComponent;
import com.almasb.fxgl.texture.Texture;
import com.almasb.fxglgames.pacman.PacmanType;
import javafx.scene.paint.Color;

public class GuardCoinComponent extends Component {
    private AStarMoveComponent astar;
    private Entity targetCoin;

    @Override
    public void onUpdate(double tpf) {
        if (targetCoin == null || !targetCoin.isActive()) {
            entity.getWorld()
                    .getRandom(PacmanType.COIN)
                    .ifPresent(coin -> {
                        highlight(coin);
                        targetCoin = coin;
                    });
        } else {
            if (astar.isAtDestination()) {
                int x = targetCoin.call("getCellX");
                int y = targetCoin.call("getCellY");
                Cell cell = astar.getGrid().get(x, y);

                astar.getGrid()
                        .getRandomCell(c -> c.getState().
                        isWalkable() && c.distance(cell) < 5)
                        .ifPresent(astar::moveToCell);
            }
        }
    }

    private void highlight(Entity coin) {
        var texture = (Texture) coin.getViewComponent().
        getChildren().get(0);
        var newTexture = texture.multiplyColor(Color.RED).
        brighter();

        coin.getViewComponent().removeChild(texture);
        coin.getViewComponent().addChild(newTexture);
    }
}
```

There are only two fields in this component: the reference to the `AStarMoveComponent` (which we are already familiar with) and the coin entity. The coin is what this AI will be guarding. Inside the `onUpdate()` method, we first check if the target coin exists and is present in the game world, otherwise picking a random coin to be our target. If the coin does exist, we simply choose a random location near the coin and move our AI to that cell. The highlighting method you see in Listing 5-6 allows us to make the selected coin stand out, so the player is aware of its position and can make gameplay decisions based on this feature. Highlighting effectively turns the coin red, as seen in Figure 5-4.

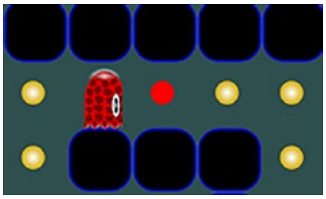


Figure 5-4. *The coin being guarded by the AI is highlighted in red*

The `DelayedChasePlayerComponent` is the next custom AI component for us to explore, whose code is in Listing 5-7.

Listing 5-7. The implementation of the player Chaser AI in our game example

```
import com.almasb.fxgl.dsl.FXGL;
import com.almasb.fxgl.entity.component.Component;
import com.almasb.fxgl.entity.component.Required;
import com.almasb.fxgl.pathfinding.astar.AStarMoveComponent;
import com.almasb.fxglgames.pacman.PacmanType;

@Required(AStarMoveComponent.class)
public class DelayedChasePlayerComponent extends Component {

    private AStarMoveComponent astar;

    private boolean isDelayed = false;

    @Override
    public void onUpdate(double tpf) {
        if (!isDelayed) {
            move();
        } else {

            // if delayed, only move when reached destination
            if (astar.isAtDestination()) {
                move();
            }
        }
    }

    private void move() {
        var player = FXGL.getGameWorld().
        getSingleton(PacmanType.PLAYER);

        int x = player.call("getCellX");
        int y = player.call("getCellY");

        astar.moveToCell(x, y);
    }

    public DelayedChasePlayerComponent withDelay() {
        isDelayed = true;
        return this;
    }
}
```

The implementation of this AI component is relatively straightforward with only two fields: the `AStarMoveComponent` reference (so the AI knows how to move) and a flag for whether the AI should move in a delayed fashion or go to the player immediately without any superficial delays. In the `move()` method, you will notice that we make the following call: `player.call("getCellX");`. It is a general-purpose shortcut that allows our entities to call methods in their attached components. For example, without this shortcut, we would have to call the full API: `player.getComponent(CellMoveComponent.class).getCellX();`. So in cases like this, where the call is unambiguous, it is possible to use the `call()` method on entities.

Our next component in Listing 5-8 shows how to implement the random palette swap for our enemy entities. While the enemies in the original game do not do this (they do turn into ghosts, but they do not constantly switch their color), it is an interesting addition to our clone, since the player is no longer able to easily figure out which enemy has which AI.

Listing 5-8. The implementation of the PaletteChangingComponent class

```
import com.almasb.fxgl.core.math.FXGLMath;
import com.almasb.fxgl.entity.component.Component;
import com.almasb.fxgl.texture.Texture;
import javafx.geometry.Rectangle2D;

public class PaletteChangingComponent extends Component {

    private Texture texture;

    private double lastX = 0;
    private double lastY = 0;

    private double timeToSwitch = 0;
    private int spriteColor = 0;

    public PaletteChangingComponent(Texture texture) {
        this.texture = texture;
    }

    @Override
    public void onAdded() {
        entity.getViewComponent().addChild(texture);
    }

    @Override
    public void onUpdate(double tpf) {
        timeToSwitch += tpf;

        if (timeToSwitch >= 5.0) {
```

```
        spriteColor = (int) (160 * FXGLMath.random(0, 2) *
        0.24);
        timeToSwitch = 0;
    }

    double dx = entity.getX() - lastX;
    double dy = entity.getY() - lastY;

    lastX = entity.getX();
    lastY = entity.getY();

    if (dx == 0 && dy == 0) {
        // didn't move
        return;
    }

    if (Math.abs(dx) > Math.abs(dy)) {
        // move was horizontal
        if (dx > 0) {
            texture.setViewport(new Rectangle2D(130*3 *
            0.24, spriteColor, 130 * 0.24, 160 * 0.24));
        } else {
            texture.setViewport(new Rectangle2D(130*2 *
            0.24, spriteColor, 130 * 0.24, 160 * 0.24));
        }
    } else {
        // move was vertical
        if (dy > 0) {
            texture.setViewport(new Rectangle2D(0,
            spriteColor, 130 * 0.24, 160 * 0.24));
        } else {
            texture.setViewport(new Rectangle2D(130 * 0.24,
            spriteColor, 130 * 0.24, 160 * 0.24));
```

```
                }
            }
        }
}
```

The implementation of `PaletteChangingComponent` is focused on choosing a random row inside the enemy sprite sheet and picking the right movement image based on which direction the entity is moving. The comments in the code show whether the move was horizontal or vertical. Once we have that information, the remaining question is whether the move was left or right (for horizontal), or up or down (for vertical). This can be determined by the difference between the current location and the previous one. If the difference is positive, then the move was to the right (for horizontal) and down (for vertical). The opposite is true if the difference is negative.

Our last component to explore is the `PlayerComponent` class. Its full implementation is available from Listing 5-9. The entire logic of the component is just about moving the player in the correct direction using the A* component. We have the four methods (up, down, left, right), which can be called from input handling code to make our player move in a specific direction. In `onUpdate()`, we check whether we can actually move in that direction and do so if that is possible. The `teleport()` method has a trivial implementation due to the flexibility of the A* grid API. The code obtains a reference to a random walkable cell and moves the player entity (or rather teleports it) directly to that cell.

Listing 5-9. The PlayerComponent class

```
import com.almasb.fxgl.entity.component.Component;
import com.almasb.fxgl.entity.component.Required;
import com.almasb.fxgl.pathfinding.CellMoveComponent;
import com.almasb.fxgl.pathfinding.astar.AStarCell;
import com.almasb.fxgl.pathfinding.astar.AStarMoveComponent;
import static com.almasb.fxglgames.pacman.components.
PlayerComponent.MoveDirection.*;

@Required(AStarMoveComponent.class)
public class PlayerComponent extends Component {

    enum MoveDirection {
        UP, RIGHT, DOWN, LEFT
    }

    private CellMoveComponent moveComponent;

    private AStarMoveComponent astar;

    private MoveDirection currentMoveDir = RIGHT;
    private MoveDirection nextMoveDir = RIGHT;

    public void up() {
        nextMoveDir = UP;
    }

    public void down() {
        nextMoveDir = DOWN;
    }

    public void left() {
        nextMoveDir = LEFT;
    }

    public void right() {
        nextMoveDir = RIGHT;
    }

    @Override
    public void onUpdate(double tpf) {
        var x = moveComponent.getCellX();
        var y = moveComponent.getCellY();

        if (x == 0 && currentMoveDir == LEFT) {
            astar.stopMovementAt(astar.getGrid().getWidth() -
            1, moveComponent.getCellY());
            return;

        } else if (x == astar.getGrid().getWidth() - 1 &&
                        currentMoveDir == RIGHT) {
            astar.stopMovementAt(0, moveComponent.getCellY());
            return;
        }

        if (astar.isMoving())
            return;

        switch (nextMoveDir) {
            case UP:
                if (astar.getGrid().getUp(x, y).filter(c ->
                c.getState().isWalkable()).isPresent())
                    currentMoveDir = nextMoveDir;
                break;
            case RIGHT:
                if (astar.getGrid().getRight(x, y).filter(c ->
                c.getState().isWalkable()).isPresent())
                    currentMoveDir = nextMoveDir;
                break;
            case DOWN:
                if (astar.getGrid().getDown(x, y).filter(c ->
                c.getState().isWalkable()).isPresent())
                    currentMoveDir = nextMoveDir;
                break;
            case LEFT:
                if (astar.getGrid().getLeft(x, y).filter(c ->
                c.getState().isWalkable()).isPresent())
                    currentMoveDir = nextMoveDir;
                break;
        }

        switch (currentMoveDir) {
            case UP:
                astar.moveToUpCell();
                break;
            case RIGHT:
                astar.moveToRightCell();
                break;
            case DOWN:
                astar.moveToDownCell();
                break;
            case LEFT:
                astar.moveToLeftCell();
                break;
        }
    }

    public void teleport() {
        astar.getGrid()
                .getRandomCell(AStarCell::isWalkable)
                .ifPresent(c -> astar.stopMovementAt(c.getX(),
                c.getY()));
    }
}
```

Now that we have covered all of the custom component classes used in our game example, it is time to turn our attention to the factory class implementation.

Building the Level and Entities

The factory implementation is in Listing 5-10. As we know, the entity factory is responsible for instantiating all entities present in our game. There are four entity types, so there are four methods annotated with `@Spawns` that create our entities. Before we cover them, we make a note of how the level is generated. It is stored as a plain text file, where each symbolic character designates an entity. For example:

```
11E1
0001
0001
P111
```

You could view this simple example as follows: “1” represents a block, “0” represents a coin, “P” represents the player, and “E” represents an enemy. The actual level is slightly bigger but follows the same convention.

We now consider each major method in Listing 5-10. The first method is `newBlock()`, which spawns a new block type entity. You will note that the `@Spawns` annotation has a “1” as its parameter. This is the same character we used in the example earlier to identify a block. Blocks are the cells that cannot be crossed and act like obstacles. In terms of the view, it is a simple JavaFX Rectangle with rounded corners. The coin entity is spawned from “0” using the `newCoin()` method and has a texture associated with it. We offset the texture so it is centered in the cell and add a bounding box. Most of these methods have been covered earlier in the book.

The construction of the player and enemy entities is a bit more complex. In the case of the player entity, we first create an animated texture from the sprite sheet. Given any texture, FXGL allows us to create an animated one quite easily using the `toAnimatedTexture()` method, which takes two parameters: the number of frames and the total animation duration. When setting the view of the player entity, we need to remember to call `loop()` so that the animation is looped indefinitely. The rest of the player construction code is simply adding relevant components, though there is one call that is of particular note:

```
new AStarMoveComponent(new LazyValue<>(() -> geto("grid"))).
```

In previous examples, you have seen how we can construct the `AStarMoveComponent` using an A* grid. However, when we are creating the player entity in this game, the grid does not exist yet, so instead we pass a `LazyValue` object with a callback to get the grid object at runtime. You will find the same approach used in constructing the enemy entity. One of the main differences for the enemy entity is how we obtain the AI component. To avoid having four different methods for creating four enemy types, we extract the AI component construction into a separate method, which cycles through the components and, each time it is called, returns a different one. This completes the overview of the factory class and its spawn methods. You are encouraged to recap material from the previous chapters with respect to the code that we did not cover in detail here.

Listing 5-10. The entity factory class implementation

```
import com.almasb.fxgl.core.util.LazyValue;
import com.almasb.fxgl.dsl.components.RandomAStarMoveComponent;
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.entity.EntityFactory;
import com.almasb.fxgl.entity.SpawnData;
import com.almasb.fxgl.entity.Spawns;
import com.almasb.fxgl.entity.component.Component;
import com.almasb.fxgl.entity.components.CollidableComponent;
import com.almasb.fxgl.pathfinding.CellMoveComponent;
import com.almasb.fxgl.pathfinding.astar.AStarMoveComponent;
import com.almasb.fxgl.physics.BoundingShape;
import com.almasb.fxgl.physics.HitBox;
import com.almasb.fxgl.texture.AnimatedTexture;
import com.almasb.fxglgames.pacman.components.
PaletteChangingComponent;
import com.almasb.fxglgames.pacman.components.PlayerComponent;
import com.almasb.fxglgames.pacman.components.ai.DelayedChasePl
ayerComponent;
import com.almasb.fxglgames.pacman.components.
ai.GuardCoinComponent;
import javafx.geometry.Point2D;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.util.Duration;
import java.util.Map;
import java.util.function.Supplier;
import static com.almasb.fxgl.dsl.FXGL.*;
import static com.almasb.fxglgames.pacman.PacmanApp.BLOCK_SIZE;
import static com.almasb.fxglgames.pacman.PacmanType.*;

public class PacmanFactory implements EntityFactory {
    @Spawns("1")
    public Entity newBlock(SpawnData data) {
        var rect = new Rectangle(38, 38, Color.BLACK);
        rect.setArcWidth(25);
        rect.setArcHeight(25);
        rect.setStrokeWidth(1);
        rect.setStroke(Color.BLUE);

        return entityBuilder(data)
                .type(BLOCK)
                .viewWithBBox(rect)
                .zIndex(-1)
                .build();
    }
```

```
@Spawns("0")
public Entity newCoin(SpawnData data) {
    var view = texture("coin.png");
    view.setTranslateX(5);
    view.setTranslateY(5);

    return entityBuilder(data)
            .type(COIN)
            .bbox(new HitBox(new Point2D(5, 5),
            BoundingShape.box(30, 30)))
            .view(view)
            .zIndex(-1)
            .with(new CollidableComponent(true))
            .with(new CellMoveComponent(BLOCK_SIZE, BLOCK_
            SIZE, 50))
            .scale(0.5, 0.5)
            .build();
}

@Spawns("P")
public Entity newPlayer(SpawnData data) {
    AnimatedTexture view = texture("player.png").
    toAnimatedTexture(2, Duration.seconds(0.33));

    return entityBuilder(data)
            .type(PLAYER)
            .bbox(new HitBox(new Point2D(4, 4),
            BoundingShape.box(32, 32)))
            .anchorFromCenter()
            .view(view.loop())
            .with(new CollidableComponent(true))
            .with(new CellMoveComponent(BLOCK_SIZE, BLOCK_
            SIZE, 200).allowRotation(true))
```

```
                // there is no grid constructed yet, so
                   pass lazily
                .with(new AStarMoveComponent(new LazyValue<>(()
                -> geto("grid"))))
                .with(new PlayerComponent())
                .rotationOrigin(35 / 2.0, 40 / 2.0)
                .build();
    }

    private Supplier<Component> aiComponents = new Supplier<>() {
        private Map<Integer, Supplier<Component>> components
        = Map.of(
                0, () -> new DelayedChasePlayerComponent().
                withDelay(),
                1, GuardCoinComponent::new,
                2, RandomAStarMoveComponent::new,
                3, DelayedChasePlayerComponent::new
        );

        private int index = 0;

        @Override
        public Component get() {
            if (index == 4) { // there are 4 enemies
                index = 0;
            }
            return components.get(index++).get();
        }
    };

    @Spawns("E")
    public Entity newEnemy(SpawnData data) {
        return entityBuilder(data)
```

```
                .type(ENEMY)
                .bbox(new HitBox(new Point2D(2, 2),
                BoundingShape.box(36, 36)))
                .anchorFromCenter()
                .with(new CollidableComponent(true))
                .with(new PaletteChangingComponent(texture("spr
                itesheet.png", 695 * 0.24, 1048 * 0.24)))
                .with(new CellMoveComponent(BLOCK_SIZE, BLOCK_
                SIZE, 125))
                // there is no grid constructed yet, so
                   pass lazily
                .with(new AStarMoveComponent(new LazyValue<>(()
                -> geto("grid"))))
                .with(aiComponents.get())
                .build();
    }
}
```

Before we proceed wiring up all parts of the game via the application class, we take a look at a simple User Interface (UI) implemented via FXML, so the player is aware of the time constraint and other in-game mechanics.

Building the UI

Writing a UI using the FXML markup is a common approach in JavaFX. Hence, FXGL provides first-class support for FXML files. Listing 5-11 has an example UI done in FXML. The example has a Pane as the root node and only three subnodes: a rectangle background for our UI and two labels, one for the score and the other to indicate the number of teleports the player can utilize.

Listing 5-11. The FXML file containing the UI definition for our game

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.shape.Rectangle?>

<Pane fx:id="root" prefWidth="80" prefHeight="840"
      xmlns="http://javafx.com/javafx/17"
      xmlns:fx="http://javafx.com/fxml/1">
    <Rectangle fill="black" width="80" height="840" />

    <Label fx:id="labelScore" translateX="10" translateY="790"
    textFill="white" />

    <Label fx:id="labelTeleport" translateX="10"
    translateY="500" textFill="white" />

</Pane>
```

In order to make use of this UI in FXGL, we need a controller class, such as the one provided in Listing 5-12. The only responsibility of the class is to bind the text of the labels we defined to their in-game properties and to attach an extra UI element – the time progress bar. Recall that the player needs to complete the game demo within a specified period of time, so our progress bar will keep the player up to date on how much time has left before the demo finishes.

Listing 5-12. The UI controller class

```
import com.almasb.fxgl.ui.ProgressBar;
import com.almasb.fxgl.ui.UIController;
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import static com.almasb.fxgl.dsl.FXGL.getUIFactoryService;
import static com.almasb.fxgl.dsl.FXGL.getip;

public class PacmanUIController implements UIController {
    @FXML
    private Pane root;

    private ProgressBar timeBar;

    @FXML
    private Label labelScore;

    @FXML
    private Label labelTeleport;

    @Override
    public void init() {
        timeBar = new ProgressBar(false);
        timeBar.setHeight(50);
        timeBar.setTranslateX(-60);
        timeBar.setTranslateY(100);
        timeBar.setRotate(-90);
        timeBar.setFill(Color.GREEN);
        timeBar.setLabelVisible(false);
        timeBar.setMaxValue(PacmanApp.TIME_PER_LEVEL);
        timeBar.setMinValue(0);
        timeBar.setCurrentValue(PacmanApp.TIME_PER_LEVEL);
        timeBar.currentValueProperty().bind(getip("time"));

        root.getChildren().addAll(timeBar);
        labelScore.setFont(getUIFactoryService().newFont(18));
        labelTeleport.setFont(getUIFactoryService().
        newFont(12));

        labelScore.textProperty().bind(getip("score").
        asString("Score:\n%d"));
        labelTeleport.textProperty().bind(getip("teleport").
        asString("Teleports:\n[%d]"));
    }
}
```

The last piece of the software "puzzle" is the `PacmanApp` class, which brings all other classes together.

Building the Application Class

Unlike the previous game example from the last chapter, this class is quite simple. This is because the game focuses on AI, components for which we already covered. In Listing 5-13, we first define our constant variables related to the time given for the level and the size of blocks, maps, and UI. These constants are then used inside `initSettings()` in a straightforward way. In `initInput()`, we simply call the player component and the five actions we had created. There are four variables in the game that are of importance, so we create them in `initGameVars()`. These are used to keep track of the player score, the number of coins left in the game, the number of teleports the player can use, and the time left to complete the level.

The `initGame()` code is of particular importance as it shows us how to parse the text file into a game world and subsequently generate an A* grid from this game world. To produce a level, we use the `AssetLoader` with the `TextLevelLoader` and provide relevant details, such as the block size. As for generating the A* grid (so that enemies can move intelligently), essentially, we call `AStarGrid.fromWorld` and provide details about the map and block sizes. We also tell FXGL what types of entities are not walkable. In this case, the `BLOCK` entity is the only entity type that is not walkable, so we use an "if" statement to signify it. In this method, we also add the grid object to the list of variables. Recall that when creating the player entity, we passed in the A* grid using a lazy value, indicating to FXGL that such an object would exist at runtime. We are now simply fulfilling that promise. Having generated the game world, we can readily compute the number of coins in the game: `getGameWorld().getEntitiesByType(COIN).size());`. Finally in this initialization method, we schedule the game timer to run each second so that the "time" variable is decremented appropriately; and we add a listener to it so that when it reaches 0, the game will be over.

As with a typical architecture of FXGL games, in `initPhysics()`, we set up collision handlers, such as the one between the player and enemies and one between the player and coins. The `initUI()` method shows a new `loadUI()` call, where we wire the FXML file and the UI controller class together, both of which we created earlier. The other methods implemented in the `PacmanApp` class are trivial.

Listing 5-13. The game application class

```
import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.entity.Entity;
import com.almasb.fxgl.entity.level.Level;
import com.almasb.fxgl.entity.level.text.TextLevelLoader;
import com.almasb.fxgl.pathfinding.CellState;
import com.almasb.fxgl.pathfinding.astar.AStarGrid;
import com.almasb.fxgl.ui.UI;
import com.almasb.fxglgames.pacman.components.PlayerComponent;
import javafx.scene.input.KeyCode;
import javafx.scene.paint.Color;
import javafx.util.Duration;
import java.util.Map;
import static com.almasb.fxgl.dsl.FXGL.*;
import static com.almasb.fxglgames.pacman.PacmanType.*;

public class PacmanApp extends GameApplication {

    public static final int BLOCK_SIZE = 40;
    public static final int MAP_SIZE = 21;
    private static final int UI_SIZE = 80;

    // seconds
    public static final int TIME_PER_LEVEL = 110;

    public Entity getPlayer() {
        return getGameWorld().getSingleton(PLAYER);
    }

    public PlayerComponent getPlayerComponent() {
        return getPlayer().getComponent(PlayerComponent.class);
    }

    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(MAP_SIZE * BLOCK_SIZE + UI_SIZE);
        settings.setHeight(MAP_SIZE * BLOCK_SIZE);
        settings.setTitle("FXGL Pac-man");
        settings.setVersion("1.0");
    }

    @Override
    protected void initInput() {
        onKey(KeyCode.W, () -> getPlayerComponent().up());
        onKey(KeyCode.S, () -> getPlayerComponent().down());
        onKey(KeyCode.A, () -> getPlayerComponent().left());
        onKey(KeyCode.D, () -> getPlayerComponent().right());

        onKeyDown(KeyCode.F, () -> {
            if (geti("teleport") > 0) {
                inc("teleport", -1);
                getPlayerComponent().teleport();
            }
        });
    }

    @Override
    protected void initGameVars(Map<String, Object> vars) {
        vars.put("score", 0);
        vars.put("coins", 0);
        vars.put("teleport", 1);
        vars.put("time", TIME_PER_LEVEL);
    }

    @Override
    protected void initGame() {
        getGameScene().setBackgroundColor(Color.DARKSLATEGREY);

        getGameWorld().addEntityFactory(new PacmanFactory());

        Level level = getAssetLoader().loadLevel("pacman_
        level0.txt", new TextLevelLoader(40, 40, ' '));
        getGameWorld().setLevel(level);

        // init the A* underlying grid and mark cells where
           blocks are as not walkable
        AStarGrid grid = AStarGrid.fromWorld(getGameWorld(),
        MAP_SIZE, MAP_SIZE, BLOCK_SIZE, BLOCK_SIZE, (type) -> {
            if (type == BLOCK)
```

```
                return CellState.NOT_WALKABLE;

            return CellState.WALKABLE;
        });

        set("grid", grid);

        // find out number of coins
        set("coins", getGameWorld().getEntitiesByType(COIN).
        size());

        run(() -> inc("time", -1), Duration.seconds(1));

        getWorldProperties().<Integer>addListener("time",
        (old, now) -> {
            if (now == 0) {
                onPlayerKilled();
            }
        });
    }

    @Override
    protected void initPhysics() {
        onCollisionBegin(PLAYER, ENEMY, (p, e) ->
        onPlayerKilled());

        onCollisionCollectible(PLAYER, COIN, c ->
        onCoinPickup());
    }

    @Override
    protected void initUI() {
        UI ui = getAssetLoader().loadUI("pacman_ui.fxml", new
        PacmanUIController());
        ui.getRoot().setTranslateX(MAP_SIZE * BLOCK_SIZE);
        getGameScene().addUI(ui);
    }

    @Override
    protected void onUpdate(double tpf) {
        if (requestNewGame) {
            requestNewGame = false;
            getGameController().startNewGame();
        }
    }

    public void onCoinPickup() {
        inc("coins", -1);
        inc("score", +50);
        inc("teleport", +1);
        if (geti("coins") == 0) {
            gameOver();
        }
    }

    private boolean requestNewGame = false;

    public void onPlayerKilled() {
        requestNewGame = true;
    }

    private void gameOver() {
        getDialogService().showMessageBox("Demo Over. Press OK
        to exit", getGameController()::exit);
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

This completes the maze action game implementation. Once you have all of the classes in place, you should be able to run the game. Observe how each AI behaves and attempts to catch, or threaten, the player in their own unique way.

Different games will place different requirements on the AI. Some will make use of the A* pathfinding algorithm as in our example, others will require writing conditional (if) statements that trick the player into thinking that the AI is learning, and there will be cases where established computer science algorithms are needed. All in all, there is no single way of addressing the AI implementation in games. Nevertheless, you now have a general understanding of the field and are aware of common implementation techniques. We are now ready to summarize the key points we covered in this chapter.

Summary

In this chapter, we covered the basics of game AI. Having AI in a game is important as it makes the game significantly more engaging for the player. More specifically, we considered how AI agents can navigate the game world using the A* pathfinding available in FXGL. The A* algorithm operates on a 2D grid or a graph, a data structure with nodes and edges, and calculates the shortest (there can be multiple) path between two nodes. Thereafter, we discussed adding behavior to AI agents. Behavior allows agents to engage with gameplay and the player to perform various actions. In FXGL, such behavior can be added by using custom components or by using a finite state machine. Finally, we put together a clone of Pac-Man with our original additions. We discussed how to implement each AI agent's behavior in the game and demonstrated how flexible FXGL can be in allowing users to write custom logic. The next chapter will focus on enhancing the visual aspects of FXGL games.

CHAPTER 6

Graphics, Visual Effects, and User Interfaces

Many visually complex features related to graphics and rendering will be discussed in this chapter. We will also cover the particle and the animation systems in FXGL in detail. In particular, we will consider

- The rendering pipeline in FXGL and abstractions over low-level drawing
- The separation between the game and the UI layers
- The particle system, its configuration, and how to use it within the game world
- How to translate, rotate, and scale game and UI elements using the FXGL API

Graphics and visual effects are a key aspect of modern games, especially so if the game is of an AAA class. While FXGL cannot match such level of graphically intensive computations, it is still able to produce impressive results. For example, a "shoot 'em up" game with numerous visual effects applied can be seen in Figure 6-1.

© Dr. Almas Baimagambetov 2022
A. Baimagambetov, *Learn JavaFX Game and App Development*,
https://doi.org/10.1007/978-1-4842-8625-8_6

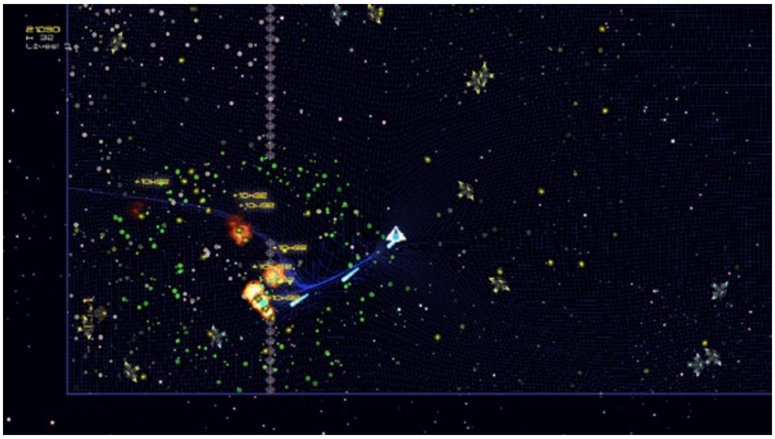


Figure 6-1. *An example top-down shooter developed with FXGL*

To get a general idea of how rendering works and what concepts are used, we open this chapter by exploring the theory behind computer graphics and how it is implemented in FXGL. In the content that follows, we will use JavaFX and FXGL interchangeably, as most of the rendering pipeline in FXGL is provided by JavaFX.

High-Level Rendering and UI

We begin our graphics journey from the very top of the rendering process, encapsulated in a platform-specific representation of a window, and we go all the way down to how each back end natively renders the elements inside the window. In JavaFX, a native operating system window is abstracted by the `Stage` class. When it comes to window rendering, commonly seen options are

- Exclusive full screen (the window gets exclusive access to the graphics adapter, making it the only thing to be rendered on the display; this approach can make it difficult to quickly switch between windows).

- Windowed (this is the typical state of any desktop application when you normally see the window decorations, including the minimize, maximize, and close buttons). This is the option that is provided by default.
- Borderless full screen (the window is maximized, and borders are removed to achieve a full screen effect; however, without exclusive access, it is much easier to switch between windows).

As you already know, inside the `Stage` object, we have a `Scene`, where all of the application gets rendered. In terms of rendering modes, in graphics, we commonly distinguish between "retained" and "immediate." In the retained mode, you typically create UI objects and set their properties, whereas in the immediate mode, you typically make graphics calls to draw some primitives. For example, suppose we wish to draw a rectangle using both approaches. Using the retained mode, we construct a new `Rectangle()` object and set its x, y, width, and height properties. Then, based on these properties, JavaFX will know how to draw the rectangle. Using the immediate mode, we would normally make four calls to `drawLine()` with appropriate parameters to draw the four sides of the rectangle. The `drawLine()` method here is used to give a contrasting example, though it should be noted that some APIs may provide a direct call to `drawRect()`, which would draw a rectangle. We now examine how these theoretical approaches relate to the JavaFX implementation, starting first with the retained mode.

Retained Mode

The retained mode of rendering in JavaFX is done via the scene graph, which is a slightly modified graph (recall that we covered graphs in the previous chapter) data structure, called a tree. A tree data structure has a single special node called the root. The scene graph root is the top-level container that drives the layout of UI objects in the scene. An example of the scene graph hierarchy can be seen in Figure 6-2, which shows how JavaFX UI objects (nodes) are organized in the scene. The scene graph is read from top to bottom and from left to right.

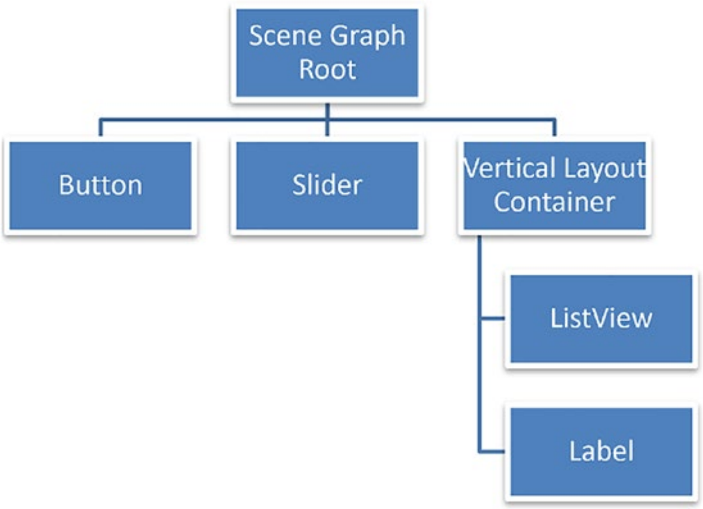


Figure 6-2. *An example JavaFX scene graph*

In the figure, you will notice that some nodes contain other nodes. For example, the vertical layout container that has `ListView` and `Label` objects is deliberately renamed to demonstrate how containers can have multiple nodes (these nodes are called children) inside them. In JavaFX, the actual name of the vertical layout node is `VBox`. Nodes that are not layout containers and are meant to be interactive from the user's perspective are commonly known as controls. These nodes do not have children. Examples of controls include `Button` and `Slider`.

The scene graph in JavaFX manages a logical representation of all UI objects active in the application. Based on this information, the scene graph can manage these objects in a very efficient way. For example, if a

UI object is not visible, then there is no need to draw it. Another example would be "partial redraw", where only a portion of the scene graph is redrawn if there were no major changes since the last frame. Generally, the scene graph retained approach is the preferred way to display your visual objects in JavaFX, and this is also true for FXGL. We should note here that all UI nodes in the visible scene can only be modified from the JavaFX Application Thread to avoid any graphical issues. In practice, this means in FXGL, you should only modify the UI from inside the `initUI()` or `onUpdate()` method. Alternatively, you can bind UI properties to FXGL in-game variables, as we have done in previous chapters, to avoid the need for manually modifying the view.

Let us consider how to use the retained rendering approach with our game world. Recall that entities themselves do not necessarily carry information regarding their visual representation since not all entities require one. In order to display an entity on the screen, we normally need to configure and attach its view via the `ViewComponent`, which then provides the scene graph with rendering data.

```
Rectangle view = new Rectangle();
// configure view as appropriate
entity.getViewComponent().addChild(view);
```

Adding this entity to the game world will automatically add it to the scene graph since it has a `ViewComponent` with an associated view inside it. In the preceding example, we used a software generated shape – a rectangle. However, most of the time you will be using textures as views. There is a `Texture` class that does exactly that:

```
Texture t = FXGL.texture("player.png");
```

It should be noted that `Texture` is also a JavaFX node, so you can use it anywhere inside the UI, not just for FXGL entities.

In the retained mode in FXGL, the scene graph is split into two: the game layer and the UI layer on top. The game layer is where your world is displayed. The coordinates in the game layer can extend beyond the visible viewport (we will cover the viewport concept a bit later in this section). As your player moves around, other parts of the world will become visible. Although most of the game layer will consist of entities, it is possible to add a view that is not associated with any entity:

```
Node node = ...
getGameScene().addGameView(new GameView(node, 0));
```

The value of 0 passed into the `GameView` constructor is the z-index for the view. The z-index is an integer value that defines the order in which views are drawn inside the game layer part of the FXGL scene graph. The higher the value, the closer the view is to the top of the drawing stack. In other words, a node with a z-index of 1 will be drawn on top of any node whose index is below 1. Similarly, if we add a node with a z-index of 2, then it will be drawn on top of the nodes with z-indices 0 and 1.

The UI layer is where your HUD (heads-up display) and various UI elements are located, including the menus. It is commonly used to provide information about the game and player states. The view is fixed in the visible viewport, and its coordinate system is always within (0, 0) to (width, height). This means that while the player can move around freely in the world, the UI stays fixed in one place. Having fixed coordinates also simplifies designing and positioning UI elements. To add a node into the UI layer, you can call the following:

```
Node node = ...
// position using setLayout / setTranslate methods
getGameScene().addUINode(node);
```

With respect to menus in the UI layer, FXGL uses the factory pattern to work with these objects. To provide your own menu implementation, there are two things we need to do: first, create your menu class that extends FXGLMenu, and second, create your factory class that extends SceneFactory. For example:

```
public class MyMenu extends FXGLMenu {
    public MyMenu(MenuType type) {
        super(type);
        getContentRoot().getChildren().addAll( ... );
    }
}
```

The MenuType.MAIN_MENU is the one shown at the start of the application. When the main menu is shown, the game has not been initialized yet. The MenuType.GAME_MENU is the one that can be opened by the player during gameplay. The view and functionality for both of these can be implemented by extending FXGLMenu. Once you have done so, we need to tell FXGL how to construct your custom menu, which is done via the scene factory:

```
public class MySceneFactory extends SceneFactory {
    public FXGLMenu newMainMenu() {
        return new MyMenu(MenuType.MAIN_MENU);
    }
    public FXGLMenu newGameMenu() {
        return new MyMenu(MenuType.GAME_MENU);
    }
}
```

Finally, the only thing that remains is telling FXGL which factory class to use. This can be accomplished by setting the scene factory inside the initSettings() method as follows:

```
settings.setSceneFactory(new MySceneFactory());.
```

Immediate Mode

Having covered basics of the retained mode in FXGL, we now briefly consider the immediate mode. If you have tried running some FXGL examples, or just skimmed through the code, you have probably noticed that there is no explicit `render()` to override in order to provide your own rendering routines. The reason for this is partially because FXGL is meant to be very high level in its API and hide most of the low-level code behind the engine curtains. The actual reason is the fact that JavaFX only gives access to the retained graphics mode, meaning we do not get to draw objects ourselves but rather say what to draw. Nevertheless, there is a `GraphicsContext` object we can use to imitate the direct drawing if there is a need for that. It can be obtained by first constructing a `Canvas` object and adding it to the UI layer as follows:

```
@Override
protected void initGame() {
    Canvas canvas = new Canvas(width, height);
    addUINode(canvas);
}
```

Next, inside the `onUpdate()` method of the application, we can get a reference to the `GraphicsContext` object and then manually draw on it.

```
@Override
protected void onUpdate(double tpf) {
    GraphicsContext g = canvas.getGraphicsContext();
}
```

We can draw on `g`, as if we are drawing directly to the screen. The graphics context, in this case, is placed on top of all elements present in the game layer (since it is added to the UI layer). Therefore, anything you draw with it will be drawn on top of existing game and entity views. It is worth noting that you should use `GraphicsContext` only if your game requires more performance or you have many entities to display; otherwise, the scene graph (retained) mode is likely to be superior.

For even greater rendering performance using the immediate mode, you can have a look at `PixelBuffer`, which was introduced in JavaFX 13 and allows drawing directly into a visible area of the screen via `WritableImage`. A useful bonus is the fact that this drawing can be parallelized. In other words, you can split the drawing area into multiple parts, such that each part is drawn by a separate thread in parallel, potentially improving the rendering performance even further. However, the major drawback of this approach is that `PixelBuffer` itself does not provide any means of rasterization. This means if you wish to draw a rectangle, you will need to draw it pixel by pixel. As you can imagine, this is a considerable limitation, and hence, the approach should only be used if you explicitly need performance. If you do wish to try out this rendering method, searching for "JavaFX `PixelBuffer` example" yields some useful results. This completes the overview of the two high-level rendering methods in FXGL, and our attention now switches to the low-level rendering capabilities and the viewport.

Low-Level Rendering and Viewport

The viewport is a window into the game world and is the visible area that the user sees. Effectively, the viewport in FXGL takes on the responsibility of a camera. When building a game world that is larger than a single screen, FXGL allows adjusting the viewport so that the "camera" can move to show other parts of the game world. You can obtain the reference to the viewport as follows:

```
Viewport viewport = getGameScene().getViewport();
```

You can change the x and y values of the viewport manually by calling

```
viewport.setX(...);
viewport.setY(...);
```

While you can use the X and Y JavaFX properties to bind to any entity's X and Y, the API has a built-in method to bind the viewport, so it automatically follows an entity:

```
Entity player = ...;
// distX, distY define how far the origin is from the entity
viewport.bindToEntity(player, distX, distY);
```

As the viewport follows the player, it may go off the level bounds. You can set the bounds to how far the viewport can "wander off":

```
viewport.setBounds(minX, minY, maxX, maxY);
```

Using this approach, you can ensure that the viewport always renders the view that is within the bounds of the game.

We have seen how to perform high-level drawing via the retained and immediate modes, including how to modify the visible area of the screen using the viewport. However, at low level, it is also important to have a rough idea of how pixels operate and what information is needed to draw a single pixel. The color of a single pixel can be represented by an RGBA object, where the letters stand for red, green, blue, and alpha, respectively. Based on how the values of each channel are represented, it is either a floating-point range between 0.0 and 1.0 or a whole number (integer) between 0 and 255. By combining the values of red, green, and blue, it is possible to construct a range of colors. The final result is also determined by the alpha channel, where 0.0 (or 0) means the color is completely transparent (effectively invisible) and 1.0 (or 255) means the color is completely opaque. Any value between this range provides some level of transparency, making it possible to see through this specific pixel. Using a combination of different transparency values, it is possible to achieve many visually appealing effects, such as the effect seen in Figure 6-3, where multiple oval shapes are overlapping and each is drawn with its own opacity value.

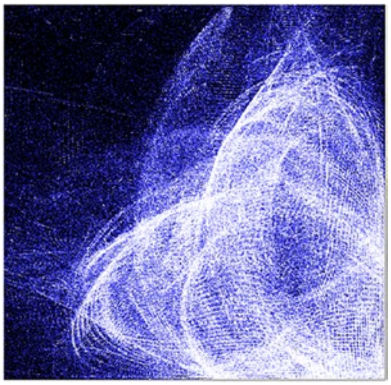


Figure 6-3. *Overlaying multiple pixels with different levels of opacity (alpha channel value)*

When pixels overlap, the final color is determined via the blending mode. Blending can be described as a function that takes two pixels, the bottom one that has already been drawn and the top one that we are about to draw, and returns the combined color. Since the function is free to implement any way of mixing the two colors to obtain the final one, there can be an infinite number of blending functions, though some (like SRC_OVER and ADD) are more common than others. For example, the default blending function that JavaFX uses is called SRC_OVER, which simply returns the top color. The ADD blending function sums the two colors and returns the result. In FXGL, the `Texture` class provides a range of these blending functions, and since they operate on the underlying image data, there is no rendering overhead. If you need dynamic blending, however, you can still use the JavaFX `BlendMode`, but it can be costly if you have a large number of nodes.

Now that we know how the final color of a pixel is processed, the final piece of the rendering stack is how to get all of this pixel information to the screen. Ultimately, these pixels will get drawn by the back-end implementation that is specific to the target platform (the platform on which the application is running). We have seen that the JavaFX pipeline provides a set of high-level APIs that can be used to achieve high-performance rendering. These APIs are implemented by a range of platform-specific classes in native code. The technologies used behind the rendering are Direct3D on Windows, OpenGL on macOS and Linux, and OpenGL ES on mobile and embedded (such as the Raspberry Pi). A visual representation of this stack is available from Figure 6-4. The modern implementation of graphics on macOS also makes use of Metal for fine-tuned performance improvements.

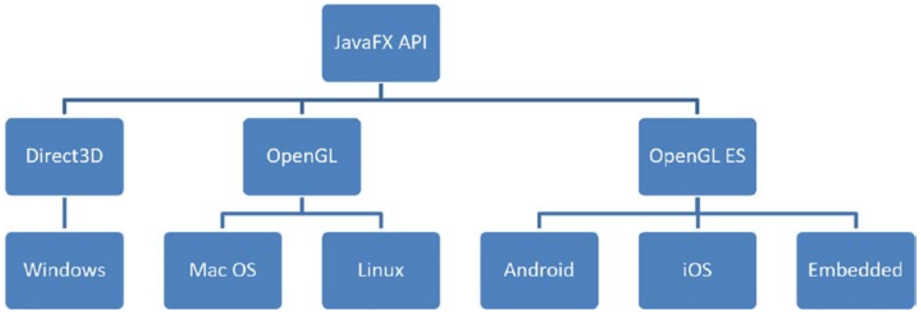


Figure 6-4. *Different JavaFX rendering back ends*

We have now completed the overview of the FXGL rendering pipeline and move on to visual effects that can be produced using this pipeline.

Visual Effects Systems

In FXGL, a number of systems exist that allow the use of visually appealing effects. All of them build on top of the aforementioned rendering stacks and provide a user-friendly API to access the required functionality. The commonly used ones are the particle, animation, and interpolation systems, which are expanded upon in the following subsections.

Particle System

Particle effects are a set of visual effects where each individual view is called a particle. These effects are typically colorful and dynamic, so they can easily bring any gameplay element to life. In FXGL (and other engines), a system that controls all particles and their behavior is called a particle system. A particle can be represented by a small geometric (possibly textured) object with certain properties, such as velocity, acceleration, scale, lifetime, opacity, and many others. Finally, a particle emitter is the construct that configures how and when particles are created. A single particle, when drawn, may appear just like any other ordinary game object. However, when drawn in large quantities, these particles can create extraordinary visual effects, such as the effect seen in Figure 6-5.

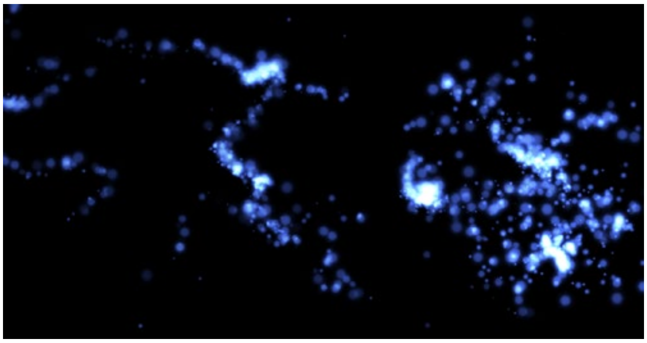


Figure 6-5. *An example of a particle effect in FXGL*

The common approach of using a particle system in FXGL involves taking a stock emitter (there are several built-in emitters provided by FXGL in the `ParticleEmitters` class) and configuring it as needed to produce the desirable visual effect. A particle emitter is not a graphical object in itself; hence, it requires a "bridge" to be added to the FXGL scene graph. There are two such bridges: `ParticleComponent` and `ParticleSystem`. As you may have guessed, the former allows attaching an emitter to an entity, thus adding the emitter to the game layer, whereas the latter attaches the emitter to the UI layer.

Configuring an emitter is generally straightforward. For example, the emitter in Listing 6-1 uses an existing explosion emitter configuration and sets the number of maximum emissions to 2, sets the number of particles per emission to 50, and, finally, sets the emission rate to 0.5 (1 means every frame; 0.5 is every two frames). In practice, the emitter will fire twice: once in the first frame and once more after skipping a frame. In each emission, there will be 50 particles. As can be seen from Listing 6-1, various other properties can be set, so the reader is encouraged to play with these settings to see their impact on the visuals.

Listing 6-1. Configuring a particle emitter

```
var emitter = ParticleEmitters.newExplosionEmitter(300);
emitter.setMaxEmissions(2);
emitter.setNumParticles(50);
emitter.setEmissionRate(0.5);
emitter.setSize(1, 24);
emitter.setScaleFunction(i -> FXGLMath.randomPoint2D());
emitter.setExpireFunction(i -> Duration.seconds(2.5));
emitter.setAccelerationFunction(() -> Point2D.ZERO);
emitter.setBlendMode(BlendMode.ADD);
// ... other properties
```

Once the emitter is configured, we can use the `ParticleComponent` to make the emitter part of the game:

```
var e = new Entity();
e.addComponent(new ParticleComponent(emitter);
```

Adding e to the game world will also automatically add the particle effect to the FXGL scene graph, thus making it visible in the game.

Similarly, to remove the particle effect from the game, all we need to do is remove the entity (to which the effect is attached) from the world.

Animation System

In addition to particle effects, there are other ways to bring any static images to life. We now cover various ways to animate visual objects. Animations are a common part of any dynamic game element or visually appealing user interface. Fundamentally, an animation is a change of some value over time. If the value we are changing directly affects any visible properties, then what we see on the screen becomes dynamic. Commonly used animations fall into one of these categories:

- Translation (changing the position of an object along the X, Y, or Z axis)
- Rotation (changing the angle of an object along the X, Y, or Z axis)
- Scale (changing the size of an object along the X, Y, or Z axis)
- Opacity (changing the transparency of an object between 0 and 100%)
- Color (changing the RGBA value of an object)
- Custom (changing the custom property of an object)

Let us consider the example in Listing 6-2. In the example, we have a standard JavaFX `Rectangle` (though you can also use an entity), which we are going to translate along the X axis. As with the entity builder that you are familiar with, we can use the animation builder to simplify the animation construction. Once we have a reference to the builder via `animationBuilder()`, we can set the duration of the single cycle, the number of times to repeat the animation, and whether it should reverse itself during alternate cycles. Next, we decide which one of the categories mentioned previously we would like to use for the animation. Since we selected translation, in Listing 6-2, we call `translate()`, which in turn allows us to configure the animation further with specific properties, such as the `from` and `to` points. Finally, we call `buildAndPlay()`, which, as the name implies, will build the animation and start playing. By default, the animation is played inside in the game scene. If you need to change the scene in which the animation is executed, for example, a menu scene, then you can simply pass the reference to your scene object when calling the animation builder. To build other animation types, appropriate methods exist, such as `rotate()`, `scale()`, `fade()`, and other more generic ones, such as `animate()`.

Listing 6-2. Animating a JavaFX node

```
var node = new Rectangle();

animationBuilder()
        .duration(Duration.seconds(2.0))
        .repeat(2)
        .autoReverse(true)
        .translate(node)
        .from(new Point2D(0.0, 100.0))
        .to(new Point2D(200.0, 100.0))
        .buildAndPlay();
```

Interpolation System

If an animation is a change of value, then interpolation (also known as easing or tweening) is the rate of this change. For example, suppose you have a fade-out animation that changes the opacity of the player entity from `1.0` to `0.0` over two seconds. By default, FXGL uses the linear interpolator, which means each second the animation will progress by the same amount. So, if it takes two seconds to go from `1.0` to `0.0`, then it is readily seen that after one second, the player's opacity will be `0.5`. If we use any nonlinear interpolator, then we will experience a different rate of the animation, so after one second, the opacity value may be greater or less than `0.5`. Visually, this means the player will fade out slower or quicker.

We should note that using different interpolators does not change the start and end values, or the duration of the animation. Instead, an interpolator will affect the values that are computed during the animation. Effectively, interpolation is a ratio between time and progress, where the linear interpolator simply provides the ratio of 1:1, that is, 50% of time passed equals to 50% of progress completed. Thus, an interpolator can readily be represented as a function:

```
double interpolate(double time) {
    double progress = ...
    return progress;
}
```

This function can convert the time value between 0 and 1 into a progress value (in the same range) using any arbitrary mathematical operation. You can see the results of common interpolators (which are already provided by FXGL) in Figure 6-6. The animation of the bird entity translates it from the left side to the right over two seconds. When the time value is 0, all bird entities, regardless of their interpolators, will start on the left, and when the time is 1, they will all finish at the same location (along the X axis) on the right. The figure shows the time value close to 0.4.

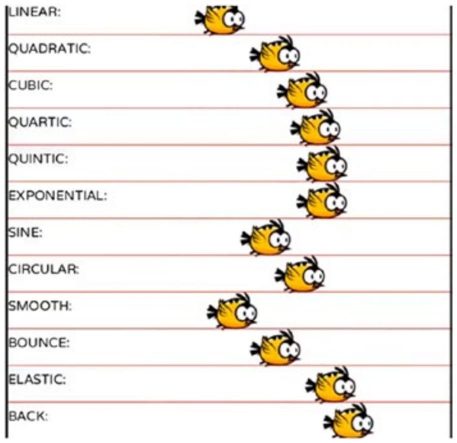


Figure 6-6. *Effects of different interpolators on the animation progress*

As a final remark, it should be noted that from the animation point of view, there is no difference between a JavaFX node and an FXGL entity. Hence, the API for both is the same. On this remark, we complete our overview of the FXGL visual systems.

Summary

In this chapter, we looked at rendering basics and how they are implemented in FXGL. We saw how to add standard JavaFX controls to the FXGL UI and how the UI layer is separated from the game layer. Using JavaFX controls an important feature as it demonstrates that JavaFX developers do not need to learn new API and can use what they know straight away. We saw how to construct complex visual effects with the particle system and configure them to fit our game. Finally, we covered how to animate game and UI elements and how to configure animations with interpolators. The key-takeaway message from this chapter is that although graphics theory can be a significantly complex topic, high-level abstractions in JavaFX and FXGL allow developers to fully utilize the rendering capabilities of the target platform without the need to access low-level implementation details.

CHAPTER 7

General-Purpose Applications and Packaging

This chapter is dedicated for general-purpose (nongame) applications that can be developed using FXGL. These include business applications, graphical editors, and visualization tools. At a high level, a game application is not very different from a front-end business application. They both run in an infinite loop until the user exits the application. They both take input, update the application state, and finally render the state to the screen. To demonstrate that FXGL can be used for developing both game and nongame applications, we will cover

- An application with a login screen and business logic
- A graph data visualizer
- A simple representation of our solar system in 3D

We will also cover the packaging and cross-platform deployment of JavaFX and FXGL applications, including games, to end users. This simplifies the process of running the end-user application and can be achieved by using the OpenJFX and the GluonFX Maven plug-ins.

© Dr. Almas Baimagambetov 2022
A. Baimagambetov, *Learn JavaFX Game and App Development*,
https://doi.org/10.1007/978-1-4842-8625-8_7

Business Application

We will begin exploring general-purpose applications with a simplistic and trivial example. The application we build takes user credentials, checks whether they are valid, and provides the API to create new accounts. You can see both the login and main user interfaces in Figure 7-1.

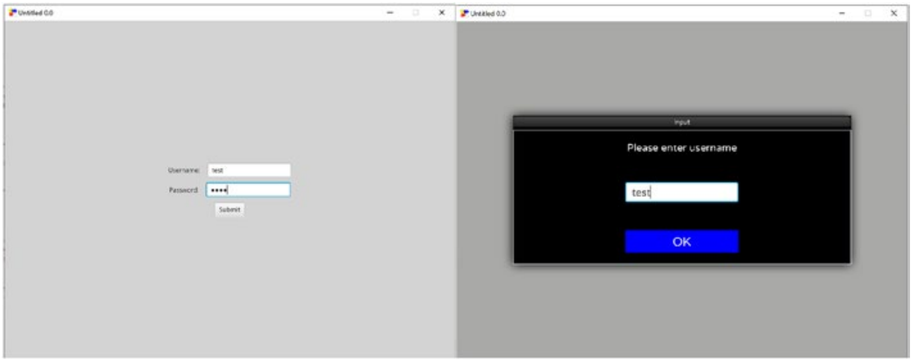


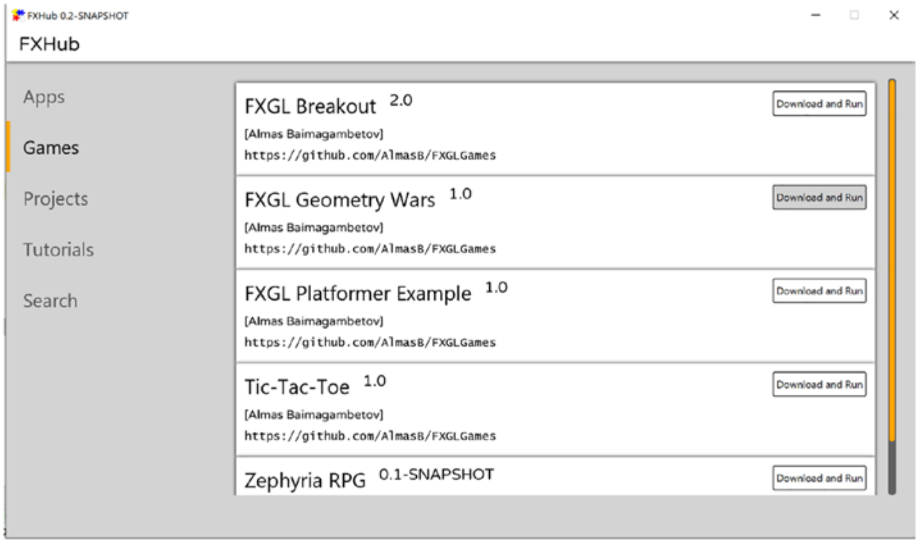
Figure 7-1. *A sketch user interface combining standard JavaFX and built-in FXGL controls*

The code of the entire application is provided in Listing 7-1. As you can see, there are two instance-level variables named `loginRoot` and `mainRoot`, which represent the login and main UI, respectively. The key part of the code is inside `initGame()`, where the majority of the application is initialized. In Figure 7-1, on the left, we observe that the UI has two fields: one for username and one for password. We create these fields in `initGame()` and wrap them with a horizontal layout box, which also contains the description label to help the user know what input they should provide. Next, we create the "Submit" button, whose action is to check if the input is valid, and if it is, we show the main view (i.e., we change the root to `mainRoot`). In a real application, we would also want to perform an action when the submitted credentials are not valid. For example, we could show a generic notification stating that the user entered invalid credentials. Finally, you will note that the `loginRoot` object is simply a vertical layout box containing the objects we created, whereas the `mainRoot` is just a container with one item – the `btnAddNew` button, which, when clicked, shows the dialog box from Figure 7-1.

Listing 7-1. An example implementation of a business application with a login screen

```
import com.almasb.fxgl.app.*;
import javafx.geometry.Pos;
import javafx.scene.Cursor;
import javafx.scene.Node;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import static com.almasb.fxgl.dsl.FXGL.*;

public class BusinessApp extends GameApplication {
    private VBox loginRoot;
    private StackPane mainRoot;

    @Override
    protected void initSettings(GameSettings settings) { }

    @Override
    protected void initGame() {
        getGameScene().setBackgroundColor(Color.LIGHTGRAY);
        getGameScene().setCursor(Cursor.DEFAULT);

        var fieldUsername = new TextField();
        var fieldPassword = new PasswordField();

        var box1 = new HBox(10, new Label("Username: "),
        fieldUsername);
        var box2 = new HBox(10, new Label("Password: "),
        fieldPassword);
        box1.setAlignment(Pos.CENTER);
        box2.setAlignment(Pos.CENTER);

        var btn = new Button("Submit");
        btn.setOnAction(e -> {
            if (isValid(fieldUsername.getText(), fieldPassword.
            getText())) {
                showMainView();
            }
        });

        loginRoot = new VBox(10, box1, box2, btn);
        loginRoot.setAlignment(Pos.CENTER);
        loginRoot.setPrefSize(getAppWidth(), getAppHeight());

        var btnAddNew = new Button("Add New");
        btnAddNew.setOnAction(e -> {
            showAddNewDialog();
        });

        mainRoot = new StackPane(btnAddNew);
        mainRoot.setPrefSize(getAppWidth(), getAppHeight());

        addUINode(loginRoot);
    }

    private boolean isValid(String name, String password) {
        return true;   // code to check if account is valid
    }

    private void showMainView() {
        removeUINode(loginRoot);
        addUINode(mainRoot);
    }

    private void showAddNewDialog() {
        getDialogService().showInputBox("Please enter name",
        name -> {
            getDialogService().showInputBox("Please enter
            password", password -> {
                // code to add new accounts
            });
        });
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

In this contrived, yet insightful, business application, we mostly used built-in JavaFX controls. However, it should be noted that FXGL user interfaces are not limited to plain JavaFX controls and that the views can be customized with CSS as with a normal JavaFX application. An example FXGL application, FXHub, with a slightly more sophisticated view, compared to the one we built, is shown in Figure 7-2. This should give you an idea of what can be achieved from the UI aesthetics point of view. We will now move on to the example in the next section that demonstrates that not all applications require a traditional user interface to be functional and useful.



FXGL Breakout 2.0
[Almas Baimagambetov]
https://github.com/AlmasB/FXGLGames

FXGL Geometry Wars 1.0
[Almas Baimagambetov]
https://github.com/AlmasB/FXGLGames

FXGL Platformer Example 1.0
[Almas Baimagambetov]
https://github.com/AlmasB/FXGLGames

Tic-Tac-Toe 1.0
[Almas Baimagambetov]
https://github.com/AlmasB/FXGLGames

Zephyria RPG 0.1-SNAPSHOT

Figure 7-2. *The main user interface for the FXHub software built with FXGL*

Graph Data Visualizer

In this section, we will use a subset of Twitter social networking data available from `https://snap.stanford.edu/data/` and develop an FXGL application to visualize this data in a form of a graph. Recall from Chapter 5 that a graph is a collection of nodes (items) and a collection of edges that connect the nodes. Graph data arises from a range of domains: from file systems and social networks to football matches and biomedical sciences. It is common to use graph visualizations to identify connections between items or to explore relationships between groups of items. A screenshot from the application we build is in Figure 7-3, where you can see a moderately complex graph.

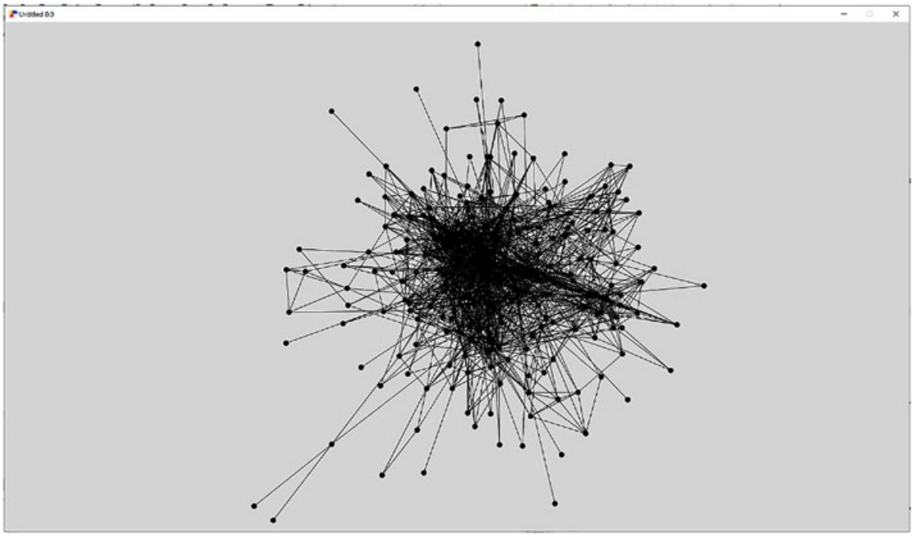


***Figure 7-3.** A visualization of an example Twitter network*

The data we obtained from the preceding source comes in a specific plain text format where each new line consists of two numbers (IDs) with a single space between them, like so:

```
number1 number2
number3 number4
...
```

Each number (ID) identifies a single node (person in our context): these are drawn as small circles in Figure 7-3. Each line of text identifies that there exists an edge (relationship) between the two nodes (people): these relationships are drawn as lines in Figure 7-3. In essence, the high-level algorithm is as follows: draw a circle for each node and draw a line for each edge. This algorithm, alongside the FXGL application, is implemented in Listing 7-2, which we will now explore in detail.

There are three fields in the `GraphVisSample` class: one to store the radius of circles that represent nodes, one to store a mapping from node

IDs to their associated entities in the application, and, lastly, one to store edges present in the graph data (this can be stored as a list if you prefer). There are also five methods of note: `initGame()`, `onUpdate()`, `makeNode()`, `forceBounds()`, and `hash()`.

In `initGame()`, we

- Load the graph data (which is in the plain text format defined previously) as a List<String>, where each item in the list is a separate line of text
- Call forEach() on the list to process each line
- Split each line into two tokens (the two int node IDs) and let makeNode() construct an entity from the node ID.
- Add a new edge between the two nodes since each line of text indicates a relationship

We see that each edge's start and end points are bound to the positions of entities that visualize the nodes. Using this approach, we do not need to manually move the edges when we move the nodes. To avoid adding an entity for each edge, we construct a single entity called "background", and we attach all edges to that entity. Finally, we store the edge data by "hashing" two node ids using `hash()`. This approach allows us to produce a unique identifier for a given pair regardless of the order in the pair.

In `onUpdate()`, we implement the layout algorithm for the graph nodes and edges based on the Fruchterman-Reingold force model. Discussions on graph layout algorithms are beyond the scope of this book; however, in simple terms, nodes that have an edge between them are attracted, whereas nodes without edges are repelled. Therefore, all we need to do is move nodes that are connected with a line closer to one another and move nonconnected nodes further away from one another. This algorithm can be achieved by iterating over each node-node pair in the application and applying the appropriate movement. We should be mindful of the fact that these movement calls may translate nodes out of the screen bounds. Typically, we do not wish for this to happen, so at the end of each check, we call `forceBounds()` that keeps all nodes within the visible part of the screen.

The `forceBounds()` method is straightforward and consists of four "if" statement checks, one for each side of the screen. If the entity (node) is beyond one of these sides, it is snapped back to stay within the screen bounds.

The `makeNode()` method constructs an entity with a `Circle` as its view and, initially, places it randomly on the screen. We also attach a built-in component called `DraggableComponent`, which we have not used before. By attaching it to any entity, we are able to click on the entity and drag it around on the screen with our mouse. Since all nodes are constantly in motion due to the graph layout algorithm, dragging one node makes the graph self-rearrange, resulting in an appealing interactive experience for the user. In fact, by adding an on-click handler that, when clicked, would bring up some information about the node, we can easily extend this application. The implementation of this feature is left up to the reader to explore.

Listing 7-2. An example implementation of a graph visualization application

```
import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.core.math.FXGLMath;
import com.almasb.fxgl.dsl.components.DraggableComponent;
import com.almasb.fxgl.entity.Entity;
import javafx.geometry.Rectangle2D;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import java.util.HashMap;
import java.util.Map;
import static com.almasb.fxgl.dsl.FXGL.*;

public class GraphVisSample extends GameApplication {
    private static final double NODE_RADIUS = 5;

    private Map<Integer, Entity> nodes = new HashMap<>();
    private Map<Integer, Boolean> edges = new HashMap<>();

    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(1600);
        settings.setHeight(900);
    }

    @Override
    protected void initGame() {
        getGameScene().setBackgroundColor(Color.LIGHTGRAY);

        Entity background = new Entity();

        getGameWorld().addEntity(background);

        getAssetLoader().loadText("twitter_edges.txt")
                        .forEach(line -> {
            var tokens = line.split(" +");

            int n1 = Integer.parseInt(tokens[0]);
            int n2 = Integer.parseInt(tokens[1]);

            Entity e1 = nodes.get(n1);
            if (e1 == null)
                e1 = makeNode(n1)

            Entity e2 = nodes.get(n2);
            if (e2 == null)
                e2 = makeNode(n2);

            var edge = new Line();
          edge.startXProperty().bind(e1.xProperty().add
          (NODE_RADIUS));
          edge.startYProperty().bind(e1.yProperty().add
          (NODE_RADIUS));
          edge.endXProperty().bind(e2.xProperty().add
          (NODE_RADIUS)
          edge.endYProperty().bind(e2.yProperty().add
          (NODE_RADIUS));

            background.getViewComponent().addChild(edge);

            edges.put(hash(e1, e2), true);
        });
    }

    @Override
    protected void onUpdate(double tpf) {
        double K = 50;

        var entities = getGameWorld().getEntities();

        // start from 1 to skip the background entity
        for (int i = 1; i < entities.size(); i++) {
            var e1 = entities.get(i);

            for (int j = i + 1; j < entities.size(); j++) {
                var e2 = entities.get(j);

                // e1 -> e2
                var v = e2.getPosition()
                               .subtract(e1.getPosition());
                double d = v.magnitude();

                // if there's an edge between e1 and e2
                if (edges.containsKey(hash(e1, e2))) {
                    double force = d * d / K * tpf;

                 e1.translate(v.normalize().multiply(force));
                 e2.translate(v.normalize().multiply(-force));
                }

                if (d > 0.0) {
                    double force = K * K / d * tpf;

                 e1.translate(v.normalize().multiply(-force));
                 e2.translate(v.normalize().multiply(force));

                    forceBounds(e1);
                    forceBounds(e2);
                }
            }
        }
    }

    private Entity makeNode(int id) {
        var e = entityBuilder()
                .at(FXGLMath.randomPoint(new Rectangle2D(0, 0,
                getAppWidth(), getAppHeight())))
        .view(new Circle(NODE_RADIUS,NODE_RADIUS,NODE_RADIUS))
        .with("id", id)
        .with(new DraggableComponent())
        .buildAndAttach();

        nodes.put(id, e);
        return e;
    }
    private void forceBounds(Entity e) {
        if (e.getX() < 0)
            e.setX(0);

        if (e.getY() < 0)
            e.setY(0);

        if (e.getX() + 10 > getAppWidth())
            e.setX(getAppWidth() - 10);

        if (e.getY() + 10 > getAppHeight())
            e.setY(getAppHeight() - 10);
    }

    private int hash(Entity e1, Entity e2) {
        var hash1 = e1.hashCode();
        var hash2 = e2.hashCode();

        return hash1 > hash2
                ? 31 * (31 + hash1) + hash2
                : 31 * (31 + hash2) + hash1;
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

You can see that with this relatively simple graph drawing application, we have already produced a piece of software that is applicable in the real world. Specifically, this demo allows us to explore graph data that arise from various sources – the data need not be from a social network. This completes our visualizer demo, while we move on to the next exciting application, combining space, 3D, and dynamic simulations.

Solar System in 3D

In this section, we will use 3D capabilities of FXGL for the first time. While this section is not meant to give a comprehensive overview of what can be done in 3D, it demonstrates basic principles of such an application in FXGL. As you progress through this section, you will note that apart from rendering, there is very little difference between 2D and 3D.

Celestial Object Data and Logic

It is likely you have already seen different representations of our solar system. One of them is in Figure 7-4, which is a screenshot from the application we will develop. Recall that when building a game, we would normally start by defining entity types. In this application, given we do not have different entity types (all of the entities shown in Figure 7-4 are just celestial objects), we instead start by defining the properties of celestial bodies. Since the objects of these bodies are predefined, that is, we will not be introducing any at runtime, it makes sense to define them using an `enum` named `CelestialBody`. The full definitions are available in Listing 7-3. In simple terms, each body has a diameter in kilometers (km), rotation speed around its axis in km per second, distance from the sun in 10^6 km, the orbital period in number of days, and finally the texture image to be used for the body. The image assets used for these bodies are obtained from `www.solarsystemscope.com/textures/` under CC-BY 4.0, though you can use any suitable images. Finally, the numeric values for bodies are available from `https://nssdc.gsfc.nasa.gov/planetary/factsheet/`.

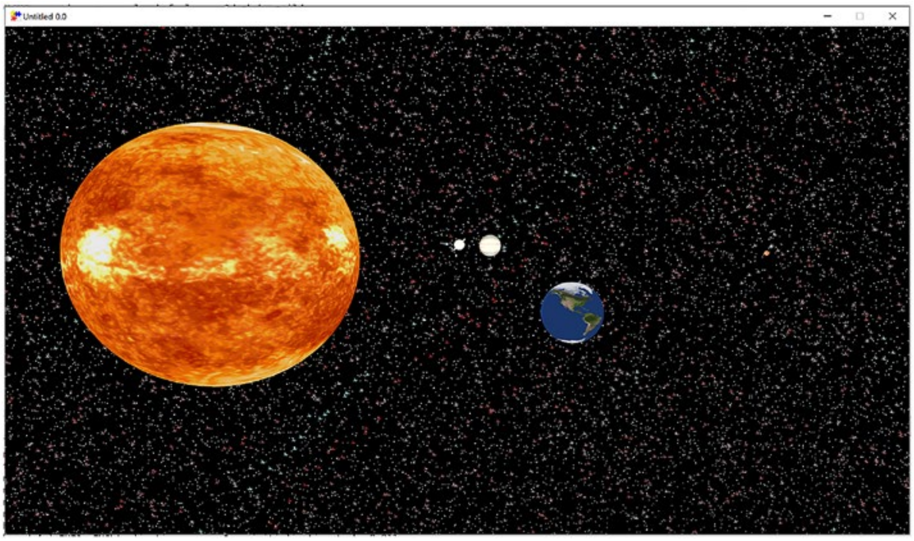


Figure 7-4. *A sketch representation of our solar system in FXGL (the sun is not scaled down accurately so that other entities can be clearly visible)*

Listing 7-3. Definitions and properties of the data used in the simulation

```
import javafx.scene.image.Image;
import static com.almasb.fxgl.dsl.FXGL.image;

public enum CelestialBody {
    // scale down with 0.2 to make others visible
    SUN(1392680 * 0.2, 1.997, 0, 0, image("2k_sun.jpg")),

    MERCURY(4879, 0.003026, 57.9, 88, image("2k_mercury.jpg")),

    VENUS(12104, 0.00181, 108.2, 224.7, image("2k_venus_
    surface.jpg")),

    EARTH(12756, 0.4651, 149.6, 365.2, image("2k_earth_
    daymap.jpg")),
MARS(6792, 0.24117, 228.0, 687.0, image("2k_mars.jpg")),

JUPITER(142984, 12.6, 778.5, 4331, image("2k_
jupiter.jpg")),

SATURN(120536, 9.87, 1432.0, 10747, image("2k_
saturn.jpg")),

URANUS(51118, 2.59, 2867.0, 30589, image("2k_uranus.jpg")),

NEPTUNE(49528, 2.68, 4515.0, 59800, image("2k_
neptune.jpg"));

// in km
private double diameter;

// in km/s
private double rotationSpeed;

// in 10^6 km
private double distanceFromSun;

// in days
private double orbitalPeriod;

private Image image;

CelestialBody(double diameter, double rotationSpeed, double
distanceFromSun, double orbitalPeriod, Image image) {
    this.diameter = diameter;
    this.rotationSpeed = rotationSpeed;
    this.distanceFromSun = distanceFromSun;
    this.orbitalPeriod = orbitalPeriod;
    this.image = image;
}
public double getDiameter() {
        return diameter;
    }

    public double getRadiusScaled(double scale) {
        return diameter * scale / 2.0;
    }

    public double getRotationSpeed() {
        return rotationSpeed;
    }

    public double getDistanceFromSun() {
        return distanceFromSun * 1_000_000;
    }

    public double getOrbitalPeriod() {
        return orbitalPeriod;
    }

    public Image getImage() {
        return image;
    }
}
```

Since all entities in the demo are celestial bodies and only need one behavior, we will use one component for all of them, whose implementation is in Listing 7-4. You will observe that it is relatively simple. Using the `enum` data for the celestial body, we build and apply two animations: one to rotate around the body axis and the other to orbit around the sun. To rotate around a specific point, we call the `origin()` method to set such a point. However, it should be noted that the first animation does not need to call the `origin()` method since, by default, the rotation in 3D is around the center of the entity, which is exactly what we need in this case. For the second animation, the rotation needs to happen around a different point, specifically around the sun entity's center. Given that we will place the sun at `(0,0,0)`, it is sufficient to set the origin to `(-entity.getX(),0,0)`, because each entity's X value equals to the distance from the world center, which is `(0,0,0)`.

Listing 7-4. The implementation of the CelestialBodyComponent class

```
import com.almasb.fxgl.entity.component.Component;
import javafx.geometry.Point3D;
import javafx.scene.shape.Sphere;
import javafx.util.Duration;
import static com.almasb.fxgl.dsl.FXGL.animationBuilder;
public class CelestialBodyComponent extends Component {
    private CelestialBody data;
    private Sphere view;

    public CelestialBodyComponent(CelestialBody data) {
        this.data = data;
    }

    @Override
    public void onAdded() {
        view = (Sphere) entity.getViewComponent().
        getChildren().get(0);

        animationBuilder()
    .duration(Duration.seconds(data.getRotationSpeed() * 3))
                .repeatInfinitely()
                .rotate(view)
                .from(new Point3D(0, 0, 0))
                .to(new Point3D(0, 360, 0))
                .buildAndPlay();
        animationBuilder()
    .duration(Duration.seconds(data.getOrbitalPeriod() * 0.1))
                .repeatInfinitely()
                .rotate(entity)
                .origin(new Point3D(-entity.getX(), 0, 0))
                .from(new Point3D(0, 0, 0))
                .to(new Point3D(0, 360, 0))
                .buildAndPlay();
    }
}
```

This completes the entire logic of this simulation. The remaining actions for us are to tell FXGL how to build entities via the factory and how to set up our 3D scene.

Celestial Object Construction

As we already know, to construct an entity, we create a factory. Listing 7-5 shows the implementation of our factory class. There is only one method since all entities are of the same type. We examine this method in detail.

Listing 7-5. The SolarSystemFactory class

```
import com.almasb.fxgl.entity.*;
import com.almasb.fxgl.scene3d.Torus;
import javafx.geometry.Point3D;
import javafx.scene.AmbientLight;
import javafx.scene.paint.*;
import javafx.scene.shape.Sphere;
import javafx.util.Duration;
import static com.almasb.fxgl.dsl.FXGL.entityBuilder;
import static com.almasb.fxglgames.CelestialBody.SATURN;
import static com.almasb.fxglgames.CelestialBody.SUN;
```

```
public class SolarSystemFactory implements EntityFactory {
    @Spawns("body")
    public Entity newBody(SpawnData data) {
        // from km to 3D units
        var sizeScale = 0.00005;
        var distanceScale = 0.00000016;

        CelestialBody bodyData = data.get("data");


        // distance sun-body, +offset by radius of each body
        var x = (bodyData.getDistanceFromSun() + SUN.
        getRadiusScaled(1.0) + bodyData.getRadiusScaled(1.0)) *
        distanceScale;

        var r = bodyData.getRadiusScaled(sizeScale);

        var mat = new PhongMaterial();
        mat.setDiffuseMap(bodyData.getImage());

        var view = new Sphere(r);
        view.setMaterial(mat);

        var e = entityBuilder(data)
                .at(x, 0, 0)
                .view(view)
                .with(new CelestialBodyComponent(bodyData))
                .build();

        if (bodyData == SATURN) {
            var torus = new Torus(r * 1.5, r * 0.3 / 3.0);
            e.getViewComponent().addChild(torus);
        }
        return e;
    }
}
```

We first define our scaling units. We do this because the actual distance and size values in kilometers are too much for our application to handle, particularly given the ratio between kilometers and 3D units is not one to one. We use different scales for distance and size so that most entities are visible in our simulation. We are not trying to achieve the scientific level of accuracy here, so such discrepancies are acceptable. Next, we obtain the body data by calling `data.get("data")`, so when spawning an entity, we must ensure to actually pass this `enum` data in. This allows us to calculate the position and size of celestial bodies in 3D units.

The views in 3D can have `Material` objects attached to them that dictate what the view looks like. We are using the default `PhongMaterial` and apply the celestial body's texture to it. Ultimately, each celestial body is represented as a sphere in our demo, so we construct a `Sphere` object using the precalculated radius, `r`, and apply the material. The construction of the entity itself is straightforward, as it is a sequence of calls that we are already familiar with. Once the entity is constructed, there is one special case for Saturn, where we add an extra `Torus` view, giving the iconic ring effect around the planet.

Building the 3D Application Class

Finally, the classes we created earlier can be brought together in the `SolarSystemApp` game application class, whose implementation is in Listing 7-6. The only instance-level field in this class is the reference to the 3D camera. In 2D, the viewport movements (which affect the 2D camera) we covered in the previous chapter were trivial, and these included translations along the X and Y axes. However, in 3D, it is much more convenient to have an abstraction over a camera, rather than the viewport.

Before we initialize and control the camera, we must first enable the 3D mode in `initSettings()`. This will make FXGL properly construct the scene and be aware that entities will now live in a 3D world, rather than 2D. In `initInput()`, the camera movements based on the keys "WASD" simply call corresponding methods in the `Camera3D` class.

Inside game initialization, we obtain the camera reference from the game scene. We then set its properties, such as the movement speed, how far the camera should see, and where it would initially be located (we translate camera Z by 10 units, so we look at the origin 0,0,0 when the camera spawns). Finally, we set the camera in FPS (first-person shooter) mode, which means when we move the mouse, the camera view will match that and rotate accordingly. We also make the cursor invisible to create a more immersive experience for the user.

Next, we add the previously created entity factory, so FXGL knows how to create bodies. The remaining methods initialize the lighting, the skybox (think a 3D background), and the bodies themselves. There will be two lights in our simulation: one that lights up the entire scene, `AmbientLight`, and the other that seemingly radiates from the sun's location, `PointLight`. Skybox is a term used to describe the collection of images drawn on the six internal sides of an imaginary cube that encloses the entire game scene in 3D. If this cube is large and far enough, then these six images create an illusion of a vast space around the camera. Skyboxes or their alternatives are commonly used in games to hide the finite borders of the game world. To create this illusion, in `initSkybox()` method, we build a 1024×1024 image and go through each pixel to color it either white or black. The color white is only given a 2% chance, so it is most likely the entire image will be largely black with some rare occurrences of white, which creates the night sky effect with stars that we saw in Figure 7-3. Finally, the creation of bodies is straightforward, since all of the logic is already encapsulated in the factory class. We go through each `enum` item for celestial bodies and use the data to spawn their respective entities.

Listing 7-6. The SolarSystemApp class

```
import com.almasb.fxgl.app.GameApplication;
import com.almasb.fxgl.app.GameSettings;
import com.almasb.fxgl.app.scene.Camera3D;
import com.almasb.fxgl.core.math.FXGLMath;
import com.almasb.fxgl.entity.SpawnData;
import com.almasb.fxgl.scene3d.SkyboxBuilder;
import com.almasb.fxgl.texture.ColoredTexture;
import com.almasb.fxgl.texture.ImagesKt;
import javafx.geometry.Point2D;
import javafx.scene.*;
import javafx.scene.image.WritableImage;
import javafx.scene.input.KeyCode;
import javafx.scene.paint.Color;
import javafx.util.Duration;
import java.util.ArrayList;
import java.util.stream.Collectors;

import static com.almasb.fxgl.dsl.FXGL.*;

public class SolarSystemApp extends GameApplication {

    private Camera3D camera3D;

    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(1280);
        settings.setHeight(720);
        settings.set3D(true);
    }

    @Override
    protected void initInput() {
        onKey(KeyCode.W, () -> camera3D.moveForward());
        onKey(KeyCode.S, () -> camera3D.moveBack());
        onKey(KeyCode.A, () -> camera3D.moveLeft());
        onKey(KeyCode.D, () -> camera3D.moveRight());
        onKey(KeyCode.L, () -> getGameController().exit());
    }

    @Override
    protected void initGame() {
        camera3D = getGameScene().getCamera3D();
        camera3D.setMoveSpeed(40);
        camera3D.getPerspectiveCamera().setFarClip(500000);
        var transform = getGameScene().getCamera3D().
        getTransform();

        transform.translateZ(-10);
        getGameScene().setFPSCamera(true);
        getGameScene().setCursorInvisible();

        getGameWorld().addEntityFactory(new
        SolarSystemFactory());


        initLight();
        initSkybox();
        initBodies();
    }

    private void initLight() {
        entityBuilder()
                .at(-5, 0, 0)
                .view(new PointLight())
                .view(new AmbientLight(Color.rgb(233, 233,
                233, 0.2)))
                .buildAndAttach();
    }

    private void initSkybox() {
        var pixels = new ColoredTexture(1024, 1024, Color.RED)
                .pixels()
                .stream()
                .map(p -> p.copy(FXGLMath.randomBoolean(0.02)
                ? Color.color(1, 1, 1, random(0.0, 1.0)) :
                Color.BLACK))
                .collect(Collectors.toList());

        Image image = ImagesKt.fromPixels(1024, 1024, pixels);

        var skybox = new SkyboxBuilder(1024)
                .front(image)
                .back(image)
                .left(image)
                .right(image)
                .top(image)
                .bot(image)
                .buildImageSkybox();

        entityBuilder()
                .view(skybox)
                .buildAndAttach();
    }

    private void initBodies() {
        for (CelestialBody body : CelestialBody.values()) {
            spawn("body",
                    new SpawnData().put("data", body)
            );
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

This simulation is now complete, allowing us to run the solar system demo. For exploratory purposes, you may wish to tweak various properties of celestial bodies to see how these changes would affect the simulation.

Packaging and Deployment

In this section, we will introduce the OpenJFX and GluonFX Maven plug-ins, using which we will be able to produce a variety of end-user packages. These include a platform-specific zero-dependency image and also a native image. We will start with the former package that allows running the packaged application without the need for the user to install Java.

To use the OpenJFX Maven plug-in with its jlink command, we typically need a `module-info.java` file, located at the sources root directory (typically `src/main/java`). The file contents are fairly trivial: we just give it a name (`mygame` in this case), and we tell it to include FXGL.

```
open module mygame {
    requires com.almasb.fxgl.all;
}
```

Our next step is to add the Maven plug-in to the appropriate (plug-ins) part of the pom.xml file, which is done in Listing 7-7.

Listing 7-7. The OpenJFX Maven plug-in

```
<plugin>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-maven-plugin</artifactId>
    <version>0.0.8</version>
    <configuration>
        <mainClass>mygame/com.mygame.GameApp</mainClass>
    </configuration>
</plugin>
```

The main class property shown in the plug-in configuration points to the class that has the `main()` method, and its name in our example is `GameApp`, located in the `com.mygame` package. We are now ready to use the command line for building zero-dependency packages. Open the operating system (or IDE) command prompt and navigate to where your pom.xml is located. Assuming the prompt recognizes the mvn command (if not, you may need to change the system settings so that the prompt knows where your Maven installation is), we simply run

```
mvn compile
mvn javafx:jlink
```

Once these commands are successfully executed, the `target` directory will contain subdirectories with binaries specific to the platform on which you are running. For example, executing the commands on Windows will produce a zero-dependency package for Windows. This means you can distribute the package among Windows users and they will not need to install Java or anything else to run it. To run the packaged Windows-specific application, you can start the .bat file located in the `bin` subdirectory, which, through a series of calls, will invoke the `main()` method.

The GluonFX Maven plug-in achieves a different goal: it performs an ahead-of-time compilation to produce a fully native package, which, unlike the previous approach, runs natively (which is not via the Java virtual machine). The simplest way to add and configure the plug-in is shown in Listing 7-8, where we use the same main class as with the OpenJFX plug-in.

Listing 7-8. The GluonFX Maven plug-in

```
<plugin>
    <groupId>com.gluonhq</groupId>
    <artifactId>gluonfx-maven-plugin</artifactId>
    <version>1.0.13</version>
    <configuration>
        <mainClass>com.mygame.GameApp</mainClass>
    </configuration>
</plugin>
```

With this plugin, there are two commands of note (other commands are beyond the scope of this book). The first one is:

```
mvn gluonfx:run
```

This command allows you to quickly test that your project successfully compiles and runs using the Java virtual machine. Once you are happy with how your project performs, we move to the second command:

```
mvn gluonfx:build
```

This is a very intensive and demanding task that converts your Java code into platform-specific native code, resulting in an executable file. For example, on Windows, this can be a .exe or .msi file. By changing the target platform, we can also build native packages for mobile and embedded devices. More information can be found at `https://github.com/gluonhq/gluonfx-maven-plugin/`.

Using both of these approaches, we end up with a set of files or packages that can be distributed to nondeveloper systems that can then execute your game or application built with JavaFX. Since an FXGL application is, for all intents and purposes, a JavaFX application, any project can also be built, packaged, and deployed in this way.

Summary

In this chapter, we explored a variety of nongame use cases for which FXGL is suitable. It is not surprising that FXGL is able to facilitate these use cases well since it is a direct extension of JavaFX. Therefore, as we once again observe, an FXGL application can do everything a JavaFX application can. In particular, we saw how we can build a general-purpose application with login and business logic, followed by graph data visualization, and a 3D simulation of our solar system. Finally, we considered how we can package and deploy our FXGL applications to make it easy for the end user to run them. In the next, closing, chapter, we recap everything we covered in this book and provide extensive external resources for reference.

CHAPTER 8

Conclusion

In this final chapter, we conclude by providing a brief summary of the chapters and giving the reader an opportunity to explore additional possibilities with FXGL. Specifically, we answer frequently asked questions, and then we discuss details of future projects that can be built on the contents of this book. Finally, there will be an extensive list of resources that can be referred to for deeper insights into FXGL.

In previous chapters, we explored what the FXGL game engine is capable of and how it seamlessly integrates with the JavaFX ecosystem. It is time to recap key-takeaway messages from each of the chapters:

- Chapter 2: We recapped Java basics in terms of classes, objects, and methods. This allowed us to construct a Java representation of real-world objects and abstract ideas. We then used the IntelliJ IDE with the Maven build tool to load JavaFX as an external library via FXGL. This approach generalizes and can be used to load any external library. We also implemented an FXGL demo.
- Chapter 3: We learned about entities and how they represent game objects. We also learned about components and how they can provide data and behavior to entities, turning a generic template into a unique game object. Entities live inside the game world and can be queried from it by a range of properties,

such as entity type, position, and the set of components it has. In this chapter, we also built our first game example – a game of Pong. Through its development, we saw how scalable the Entity-Component model is in games.

- Chapter 4: We covered physics basics and learned about various ways to detect collisions. Collision detection is a key aspect of any game since it enables interaction between game objects. We saw how FXGL allows developers to add collision handling code and modify properties of physics-driven entities. Using these physics concepts, we developed a platformer game where the player must reach the end of the level by interacting with entities.
- Chapter 5: We learned a little bit about graph theory, the A* pathfinding algorithm, and how searching a graph allows us to find paths between two points in a game level. In FXGL, this functionality is provided by the `AStarMoveComponent`, which uses level data to navigate the game world. This chapter also focused on AI and how developers can add informed behavior to entities via either custom components or the finite-state-machine implementation. Finally, we developed a maze action game, where each enemy has a distinct set of AI behaviors.
- Chapter 6: We covered the basics of graphics theory and how it translates to the FXGL (JavaFX) rendering pipeline. We configured a particle emitter to produce custom visual effects. We saw how JavaFX UI objects can be used within the FXGL environment. This is a

significant advantage of FXGL since there is no need to learn any new API for generating complex user interfaces. To bring these interfaces to life, we explored the animation system and built-in interpolators, which can change how the animation looks overall.

- Chapter 7: We saw how game and business applications have a lot in common when it comes to implementation. We developed simple nongame applications using FXGL, including a login screen with business logic, a graph data visualizer, and an animated 3D solar system model. Finally, we explored how to use Maven plug-ins to package FXGL apps.

Having completed these chapters, you are well suited to continue your game or application development journey with FXGL 17. The following sections provide extra resources to help you achieve your goal more effectively.

Frequently Asked Questions

This section covers a variety of questions related to FXGL, including its API, software architecture, and development, that are common in the community.

How to change global settings? FXGL offers a lot of built-in settings that can be easily configured from the `initSettings()` method using the `GameSettings` object. Here are some important examples:

- setApplicationMode(ApplicationMode.DEBUG) – Enables all debug logging calls.
- setScaleOnResize(boolean) – If true, the game contents will be automatically scaled up or down during resize depending on the user device resolution.

- setFullscreenAllowedFromStart(boolean – If true, allows the game to enter full screen immediately upon start.

How to change audio volume? The audio volume, both for sound effects and music, can be changed by accessing the runtime settings. Note that these are not identical to the settings object from the `initSettings()` method, though a lot of the settings overlap. To get the runtime settings and change the volume, you can call:

```
getSettings().setGlobalSoundVolume(0.5);
getSettings().setGlobalMusicVolume(0.5);
```

The following questions focus on the general development of FXGL and the framework as a whole, rather than code that achieves a specific functionality.

What was the reason for developing FXGL? It started off as a set of custom Java classes on top of the existing JavaFX functionality used in lectures on basics of game development. As the code base grew and became flexible, I continued to use it to help me run JavaFX tutorials on YouTube so that I could avoid typing the same code across tutorials. As I iterated over the code and its API, eventually, these simple classes formed packages and, later, modules. Hence, I decided to make a library out of it and share with everyone who is interested in learning JavaFX and (or) game development.

I am doing a school (academic) project, can I use FXGL? If you are a student, you should address this question to your tutor to find out if third-party libraries are allowed in your project. If you are the tutor, then feel free to contact me if you need help adapting your course to use FXGL. There are a number of academic institutions around the world who already make use of FXGL in their curriculum.

If I use FXGL for a project, do I need to worry about license or credits? No. The FXGL code base is provided under the MIT license, so feel free to do anything you want with it. Something like "Powered by FXGL" would be nice, but it is up to you. FXGL is suitable for academic, hobby, and commercial projects.

Will FXGL help my employability? If you are serious about game development and planning a future career in this field, working for an AAA company, I would suggest that you use Unity (C#) or Unreal Engine (C++), both of which are available for free. They are the engines that are widely recognized in the industry. However, if you wish to become an independent game developer, working for a smaller company, and would like to continue using Java, then all FXGL, libGDX, and JMonkey are good alternatives depending on your use case.

What were the key points learned during FXGL development? The FXGL engine has been (and still very much is) in development since 2015. The entire development timeline can be split into two: the early stage and the late stage. In the early stage, in terms of software architecture and design, everything is tentative. Designing the engine from top-down has been useful to identify significant areas of focus, such as gameplay, networking, file system access, and other subsystems.

As for implementation, during the first several versions, you just play around with the system and the code. The goal here is to identify what works and what does not both in terms of API and functionality. It is fine to write throw-away code to prototype quickly at this point to test out functionality. However, developers should be mindful that the code will be thrown away, so no major decisions should be made to avoid code rigidity.

The early stages of development are also good for establishing technical limitations of the chosen software (and hardware) stack. For example, one limitation of the JavaFX rendering pipeline is that it does not support external shaders (GPU programs that can change how something is drawn). This limitation restricts how creative developers can get when designing the visuals of their game.

In the late stages of development, once you get to know the API and how users utilize the system, you can start focusing on those core bits and build the rest of the functionality around it. It is also good to start writing your tests for the core bits so you know what they are supposed to do. In particular, you can start putting together a formal specification for your engine, and via tests, you can show that it actually does that. It is important to specify exactly what the engine is trying to achieve to avoid any distractions and features that do not need implementation. Ultimately, this saves time and improves the quality of the core items.

It should be noted that at this (late) stage, the development should be steady but slow. Growing the code too fast may lead to poor design decisions, ultimately making the code less extensible. Engine users should be consulted frequently to ensure that the functionality being implemented is actually being used. Involving engine users in the development process is a good way to identify the core subsystems that are important to them.

There are various low-level and high-level tips and tricks learned during the development. However, one that stands out is the use of an event bus, which can help decouple various engine systems. Suppose we have two so-called systems: achievements and audio. To ensure relatively easy code maintenance, we would like to ensure that one system does not have a dependency on the other (i.e., the achievements system does not call any methods from the audio system and vice versa). If there are no dependencies, then we can readily make a change in one system while not affecting the code base of the other system. Now, suppose our problem is playing a sound effect (a function in the audio system) every time an achievement has been unlocked (a function in the achievements system).

To address the preceding problem while ensuring there are no dependencies between the two systems, we can use an event bus. Somewhere in our code base there is an overarching system, which we call the engine, that maintains other systems. In the engine system, we can have the following pseudocode:

```
onAchievement(eventData) {
    audioSystem.playSound( ... );
}
// for conciseness we are passing a function as an object
eventBus.addEventHandler(EventType.ACHIEVEMENT, onAchievement)
```

When there is a new unlocked achievement, the achievements system fires an event of type EventType.ACHIEVEMENT. The event is caught by the event bus and is passed to the engine system, which, in turn, uses the audio system to play a sound. Using the described approach, changes in the achievements and the audio systems do not affect one another, though may affect (i.e., result in a public API change) the engine system. The utilization of the event bus in this way was one of the significant architectural patterns that were learned during the development and helped build FXGL's robust infrastructure.

Future Projects and Extra Resources

We have seen a number of different projects that can be developed with FXGL. In this section, we explore what further ideas can be turned into demonstrable prototypes. These ideas include

- Hand tracking visualization: https://github.com/AlmasB/HandTrackingFXGL,
- Open-world role-playing game (RPG): https://github.com/AlmasB/zephyria, and
- High-performance rendering: https://github.com/AlmasB/FXGL-FastRender.

We will now briefly consider each project in the preceding order.

To develop a hand tracking visualization tool, we first need to obtain the data. The MediaPipe library provides access to the hand tracking data processed by a Machine Learning application. There are several ways to make use of the library, with the easiest approach for this case being the JavaScript version, which is only a few lines of code. The key process is to capture the data from JavaScript and send it to our FXGL application, which can be done using web sockets. Luckily, there are libraries available for web sockets, meaning we only need to properly set them up and we can stream the hand tracking data from the MediaPipe library into our tool. An example of what such a visualization tool might produce is in Figure 8-1. You are welcome to explore the repository and turn this demo into a usable piece of software.

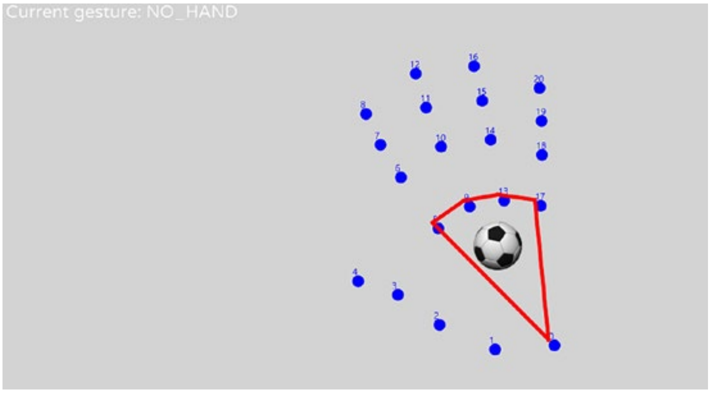


Figure 8-1. *A sketch for an FXGL hand tracking data visualization tool*

Zephyria is an open-world RPG game written completely in the Kotlin programming language. The fact that it uses a language other than Java is significant, as it demonstrates that Kotlin users can also access all of FXGL's capabilities. The game features large open-world maps covering up to 250000 tiles, where each tile is a square with 32 pixels in size. The map is loaded from a Tiled level file and is traversed using the A* algorithm we covered in Chapter 5. A screenshot from the game is in Figure 8-2.

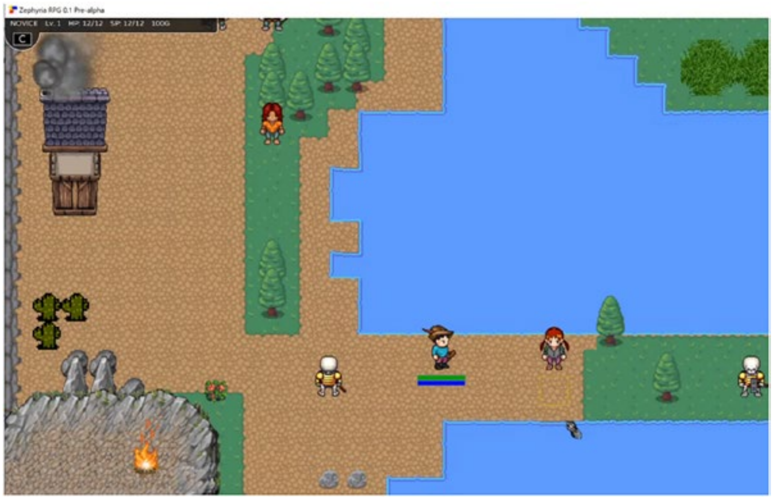

Zephyria RPG 0.1 Pre-alpha
NOVICE Lv. 1 HP 12/12 SP 12/12 100G
C

Figure 8-2. *An example map in Zephyria RPG*

In Chapter 6, we discussed the immediate and retained modes of rendering and how the immediate mode can improve performance. It is possible to yield further performance gains if we combine the immediate mode with GPU parallel computing. Effectively, the idea is that both the logic and rendering are configured in such a way that each computation can be run in parallel. If that precondition is true and the computation units are straightforward enough to translate to GPU commands, then running such an algorithm on GPU is likely to significantly boost performance. For example, Figure 8-3 shows a particle swarm rendered with FXGL that includes 10 million particles. The reader is encouraged to browse the repository link mentioned previously to see how various rendering approaches can either improve or hinder performance.

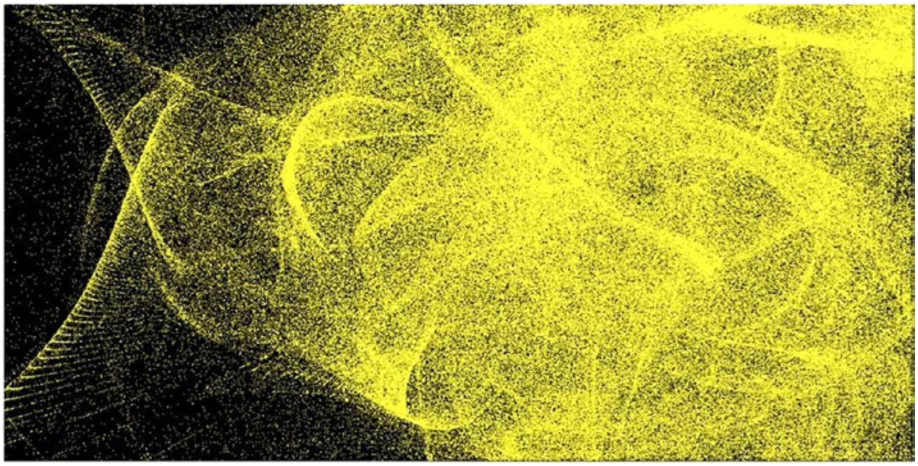


Figure 8-3. *A particle swarm example using FXGL*

We now provide useful resources for further research and development:

- FXGLGames: `https://github.com/AlmasB/FXGLGames` –This is a collection of pre-built FXGL templates and example demos, including full games. Contributions are welcome to the repository for games that are still in development.
- Online resources: `www.youtube.com/c/AlmasB0/videos`, `https://github.com/AlmasB/FXGL/wiki` – These are constantly updated to ensure they represent the latest FXGL version.
- Community articles and games: `https://github.com/AlmasB/FXGL#community` – There is a steadily growing community of FXGL users, whether beginners, students, hobbyists, or professionals. A number of articles have been written by the community to demonstrate various FXGL features, including some sophisticated examples.

Finally, if you are feeling overwhelmed, you can always join the FXGL discussion area at `https://github.com/AlmasB/FXGL/discussions`, where you can ask questions specific to your use case.

Concluding Remarks

If you have reached this point, then I would like to thank you for your perseverance and for completing your journey with FXGL 17. I sincerely hope the book has been helpful in getting to know what JavaFX and FXGL are capable of within the context of game and application development. While this book only covered the tip of the iceberg, you should now have a solid foundation on which you can continue building, and the aforementioned resources can help you do just that. I wish you the very best in all your future endeavors and look forward to seeing your projects!

Index

© Dr. Almas Baimagambetov 2022
A. Baimagambetov, *Learn JavaFX Game and App Development*,
https://doi.org/10.1007/978-1-4842-8625-8