Community Experience Distilled

Slick2D Game Development

Develop simple, yet engaging games with the Slick2D game engine

Jacob Bevilacqua

[PACKT] open source* community experience distilled PUBLISHING

Slick2D Game Development

Develop simple, yet engaging games with the Slick2D game engine

Jacob Bevilacqua

[PACKT] open source
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

Slick2D Game Development

First published: October 2013

Production Reference: 1211013

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

ISBN 978-1-78328-983-7

www.packtpub.com

Cover Image by Jarek Blaminsky (milak6@wp.pl)

Credits

Author
Jacob Bevilacqua

Reviewers
Jeff Lunt
Kyle Newton
Ernest Pazera

Acquisition Editor
Pramila Balan
James Jones

Commissioning Editor
Govindan K

Technical Editors
Jinesh Kampani
Aman Preet Singh

Project Coordinator
Michelle Quadros

Proofreader
Kevin McGowan

Indexer
Tejal Daruwale

Graphics
Disha Haria

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

Jacob Bevilacqua is a student who works on open source projects as well as game development projects in his free time. He has worked on several open source projects and has started one himself. Jacob is proficient in Java and Java game development, using both Slick2D and LibGDX.

A big thanks to my family for helping me fix spelling and grammar mistakes. I couldn't have done it without them.

About the Reviewers

Jeff Lunt got a start in programming as a kid, hacking away on his family's second-hand Apple computer, desperately trying to read and understand any code that was available to him. He pursued game programming as a serious hobby for over 10 years, using a number of languages and tools with Apple, MS DOS, MS Windows, and OS X. He's now happily living in the world of Debian-flavored Linux most of the time.

Jeff currently makes his living as a web developer of clinical research tools for Northwestern University's Feinberg School of Medicine in Chicago.

Kyle Newton is a games enthusiast. When he can't find time to make them, he's usually reading or talking about them or, on the rare occasion, actually playing them. He has a small family and is looking forward to teaching his children the joys of playing. More information about Kyle can be found at his blog `http://n3wt0n.com/` and his games at `http://waggsoft.com/`.

Ernest Pazera lives and works as a senior web developer in southeast Wisconsin, and has dabbled in game development since he was thirteen, which is one third of his age at the time of writing.

He is the author of *Isometric Game Programming with DirectX 7.0* (published by Muska & Lipman, 2001), *Game Developer's Guide to Cybiko* (published by Wordware Publishing, Inc., 2001), *Focus On 2D in Direct3D* (published by Muska & Lipman, 2002), and *Focus on SDL* (published by Muska & Lipman, 2003). He also served as a contributing author on Beginning Game Programming (published by Cengage Learning, 2009), and was the Technical Reviewer for *RPG Game Programming* (published by Muska & Lipman, 2002).

I would like to acknowledge Kevin Glass, the creator of Slick 2D, and congratulate Kevin.

Table of Contents

Preface

Thanks to the group of great community members, Slick2D has become a simple and useful tool in game development. Slick2D is an open source project run completely by the community.

Slick2D allows people from every game development background, including programmers, artists, and game designers, to easily and efficiently, create great games no matter what their skill level is. This book will help you get started in the wonderful world of game development.

What this book covers

Chapter 1, *Setting Up Slick2D*, covers how to download all the necessary libraries and files for Slick2D programming.

Chapter 2, *Game Structure*, introduces the two types of game structures Slick2D allows us to use. We will also cover how to set up a game using both game structures.

Chapter 3, *The Slick2D Workflow*, explains all of the tasks included in the Slick2D workflow and how we can set them up.

Chapter 4, *A Deeper Look at Rendering*, covers in-depth how to render an array of different things to the screen, including primitive shapes and images.

Chapter 5, *A Look at Input*, explains how we can utilize user input to bring games to life. We will cover how to incorporate both keyboard and mouse input into games.

Chapter 6, *Sound and Music*, introduces the techniques we can use to play both sound and music in our games.

Chapter 7, *From Example to Game*, covers all the things we need to do to bring the game we will work on throughout the book to life.

Appendix A, Full Source Code, includes the completed source code for the game we create in the book. This should be used for reference only, you should try to complete the game without referencing the book if possible.

Appendix B, Packaging our Game, covers the steps to take when we are ready to package a Slick2D game for distribution.

What you need for this book

To work along with the examples in this book and to create the game throughout the book, it is necessary to have a current version of the Java JDK available at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

You will also need the Slick2D and LWJGL library and native files. We will go over how to obtain these in *Chapter 1, Setting Up Slick2D*.

It is also recommended that you have the IDE Eclipse but it is not necessary and any other IDE, or even a notepad if you want, but I will explain the set-up of Slick2D and usage of Eclipse.

You can download eclipse here at `http://www.eclipse.org/`.

Who this book is for

This book is for anyone who has a basic understanding of Java and Java syntax but wants to take on game development.

You don't need to have any previous game development experience to read this book.

Conventions

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

Code words in text are shown as follows: "Add the `BasicGame` constructor inside the class and `super`, which is a string containing the title of your game.".

A block of code is set as follows:

```
public OurFirstGame() {
  super ("YOUR TITLE GOES HERE");
}
```

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

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `http://www.packtpub.com/sites/default/files/downloads/9837OS_graphics.pdf`

Errata

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

Piracy

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

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

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

Questions

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

1 Setting Up Slick2D

In this chapter we will cover the following setup tasks.

- What is Slick2D?
- Setting up the jar files
- Handling native files
- Setting up a project in eclipse

What is Slick2D?

Slick2D is a multi-platform library for two dimensional game development that sits upon the **LWJGL** (**Light-Weight Java Game Library**). Slick2D simplifies the processes of game development such as game loop, rendering, updating, frame setup, and state-based game creation. It also offers some features that LWJGL does not, such as particle emitters and integration with **Tiled** (a map editor).

Developers of all skill levels can enjoy Slick2D, as it offers a degree of simplicity that you can't find in most libraries. This simplicity not only makes it a great library for programmers but artists as well, who may not have the technical knowledge to create games in other libraries.

Downloading the Slick2D and LWJGL files

The Slick2D and LWJGL jar files, plus the LWJGL native files, are needed to create a Slick2D game project. The only system requirement for Slick2D is a Java JDK. To get the files, we perform the following steps:

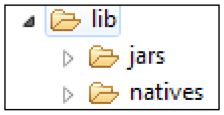
1. Obtaining the LWJGL files:
    1. Navigate to `http://www.lwjgl.org/download.php`.
    2. Download the most recent stable build.
    3. The `.zip` file will include both the LWJGL jar file and the native files. (This .zip file will be referenced as `lwjgl.zip` file later in this chapter.)
2. Obtaining the Slick2D files:
    1. Due to hosting issues, the Slick2D files are being hosted by a community member at `http://slick.ninjacave.com`. If this site is not available, follow the alternative instructions at step 3.
    2. Click on **Download**.
3. Alternative method of obtaining the Slick2D files:
    1. Navigate to `https://bitbucket.org/kevglass/slick`.
    2. Download the source.
    3. Build the ant script located at `slick/trunk/Slick/build.xml`
    4. Build it in eclipse or command line using `$ ant`.

Setting up an eclipse project

We will utilize the Eclipse IDE that can be found at `http://www.eclipse.org/` when working with Slick2D in this book. You may, however, utilize other options.

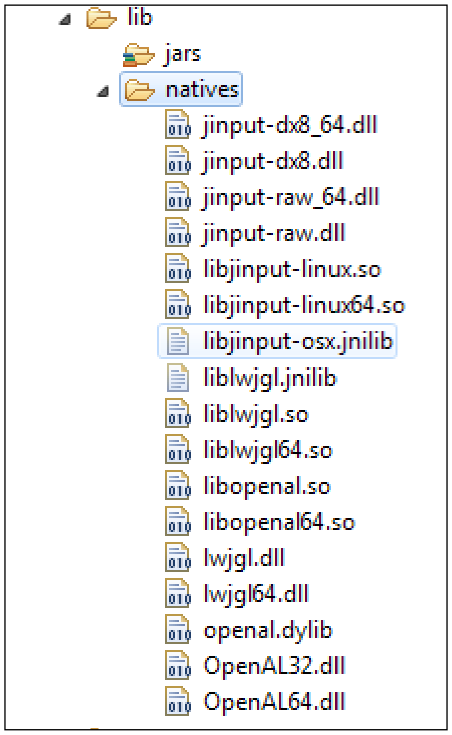
Perform the following these steps to set up a Slick2D project:

1. Navigate to **File** | **New** | **Java Project**.
2. Name your project and click on **Finish**.
3. Create a new folder in your project and name it `lib`.
4. Add two subfolders named `jars` and `native`.



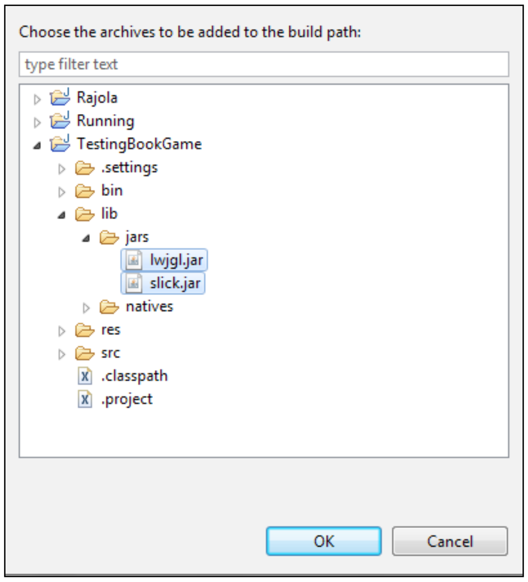
5. Place both `lwjgl.jar` and `slick.jar` in the `jars` subfolder inside our eclipse project.
6. Take all the native files from `lwjgl.zip` and place them in the `native` subfolder.

[Copy the contents of the subfolders inside native from `lwjgl.zip` not the subfolders themselves.]



7. Right-click on **project** then click on **Properties**.
8. Click on **Java Build Path** and navigate to the **Libraries** tab.

9. Add both the jars from the project.

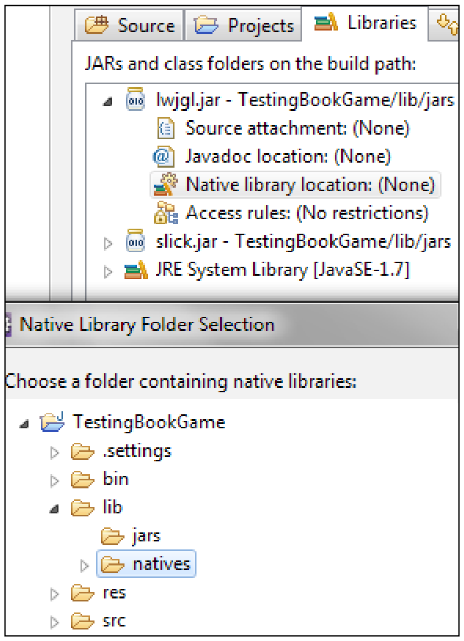


Choose the archives to be added to the build path:

type filter text

- Rajola
- Running
- TestingBookGame
  - .settings
  - bin
  - lib
    - jars
      - lwjgl.jar
      - slick.jar
    - natives
  - res
  - src
  - .classpath
  - .project

10. Select and expand `lwjgl.jar` from the **Libraries** tab and click on **Native library location: (None)** then click on **Edit** and search the workspace for the native's folder.



JARs and class folders on the build path:

- lwjgl.jar - TestingBookGame/lib/jars
  - Source attachment: (None)
  - Javadoc location: (None)
  - Native library location: (None)
  - Access rules: (No restrictions)
- slick.jar - TestingBookGame/lib/jars
- JRE System Library [JavaSE-1.7]

Native Library Folder Selection

Choose a folder containing native libraries:

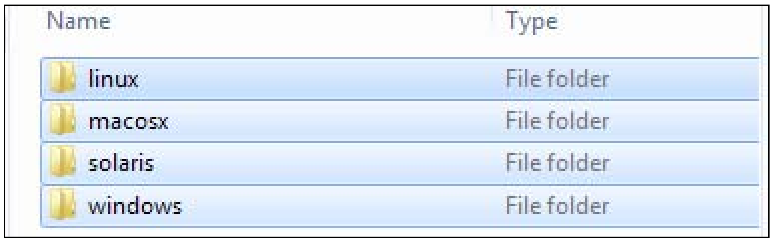
- TestingBookGame
  - .settings
  - bin
  - lib
    - jars
    - natives
  - res
  - src

Native files

The native files included in `lwjgl.zip` are platform-specific libraries that allow the developers to make one game that will work on all of the different platforms.

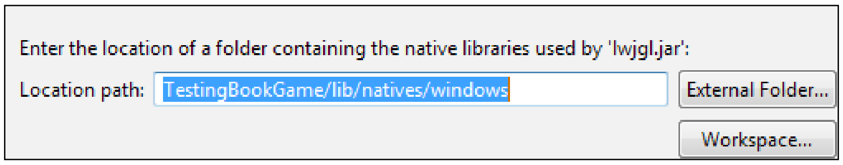
What if I want my game to be platform-specific?

No real benefit exists to being platform-specific with Slick2D. In the foregoing tutorial, we will establish a game as a multi-platform game. However, if you want your game to be platform-specific, you can make it platform-specific. In the previous tutorial (step 6) we took the content of each operating system's folder and put that content into our native folder. If, instead, you desire to make your game platform-specific, then instead of copying the contents of these folders, you would copy the entire folder as illustrated as follows:



Name	Type
linux	File folder
macosx	File folder
solaris	File folder
windows	File folder

When defining the natives for LWJGL (step 10 in previous example), simply point towards the operating system of your choice.



Enter the location of a folder containing the native libraries used by 'lwjgl.jar':

Location path: TestingBookGame/lib/natives/windows

Summary

The first chapter of any technical book is a necessary evil. We learned tons of important things necessary to create a project in Slick2D. So far we covered:

- Downloading the necessary library files
- Setting up a project (platform-specific or multi-platform)
- Native files

2
Game Structure

In this chapter we will cover the following topics:

- Basic game structure
- State-based game structure
- Setting up the game container

Game structures

In Slick2D there are two main structures, the **basic game** and the **state-based game**. Either of the structures can be used in a project. Generally, a smaller game uses the basic game structure while larger games typically use the state-based game structure. However, this rule is not set in stone and the use of one or the other structure will ultimately depend upon the requirements of the specific project. Let's take a look at each game structure.

The basic game structure

The basic game structure of a Slick2D game includes the constructor, **Slick2D workflow**, and the main method that can be found in all the Java programs. We will talk more about the Slick2D workflow in *Chapter 3*, *The Slick2D Workflow*. The basic game structure does not provide the option to move between screens or states as they are called in Slick2D.

The first step to creating a basic game

The first step to creating a basic game is to create the skeleton of a game.

Create a new class and name it what you like.

```
public class OurFirstGame {}
```

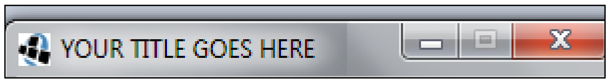
Extend the class with `BasicGame` and import `BasicGame`.

```
import org.newdawn.slick.BasicGame;
public class OurFirstGame extends BasicGame {}
```

Add the `BasicGame` constructor inside the class, and `super`, which is a string containing the title of your game.

```
public OurFirstGame() {
  super ("YOUR TITLE GOES HERE");
}
```

The title of your game will appear on the top left corner of the game window.



Add the `main` method. We will learn all of the things we can add to the main method later in this chapter.

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

Next we need to add the Slick2D workflow

We will go over the Slick2D workflow in depth in *Chapter 3, The Slick2D Workflow*, for now we will just add the necessary methods inside the `OurFirstGame` class.

Add the `init` method and add a `throws` declaration for `SlickException`:

```
public void init(GameContainer gc) throws SlickException {}
```

Add the `render` method and add a `throws` declaration for `SlickException`:

```
public void render(GameContainer gc, Graphics g) throws SlickException
{}
```

Add the `update` method and add a `throws` declaration for `SlickException`:

```
public void update(GameContainer gc, int delta) throws SlickException
{}
```

Ensure that all methods throw a `SlickException` exception. Alternatively you may use a try catch inside any of the methods.

```
public void init(GameContainer gc) {
  try {
  } catch (SlickException e) {
  e.printStackTrace();
  }
}
```

Next we need to complete the main method

We will take a deeper look at all the options you have when creating a window later in this chapter, for now we will add just the essentials.

A basic setup for the main method would look as follows:

```
public static void main(String[] args) {
  try {
    AppGameContainer agc = new AppGameContainer(new
      OurFirstGame());
    agc.setDisplayMode(400,400,false);
    agc.start();
  } catch(SlickException e) {
    e.printStackTrace()
  }
}
```

[Notice the try-catch statement around all of the code inside of the main method.]

Let's break down the main method

The main method is a crucial part of the game because it sets up the window and starts the workflow. Following are the essential parts of any main method in Slick2D.

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

The first line we already created when we made the skeleton for our basic game. It should look familiar as it is a necessary method in all the Java applications.

```
AppGameContainer agc = new AppGameContainer(new OurFirstGame());
```

The second line creates a new instance of an `AppGameContainer`. The app game container is essentially the window of the game. When creating the app game container we set up certain variables that alter the look of the window. The app game container takes one parameter; a Game interface. We should create a new instance of our class because `BasicGame` uses the interface Game.

```
    agc.setDisplayMode(400,400,false);
```

The third line defines the display mode for the game. In other words we are defining the size of the window that our game will be in. The first parameter is the width of the window in pixels. The second parameter is the height of the window in pixels. The third parameter is a Boolean value for full screen. If we wanted the game to be full screen we would write true for the final parameter.

```
    agc.start();
```

The fourth line is debatably the most important. Even if we set up the game perfectly, without any errors, and forget this line, the game won't work. `start()` takes all the variables we have set and uses them to actually create the game and begins the game loop.

`start()` should be the last method called inside the main method.

There are several other variables we can alter in the app game container. We will go over all these later in the chapter.

State-based games

A state-based game is a structure where the game switches between states during runtime. The game may switch between these states and may return to a state that has already been visited.

When revisiting a state it will be re-initiated and all variables will be reset.

The uses of a state-based game

Using a state-based game is a great idea if your game requires many different screens. These screens could be anything from a menu to the playable area of the game. In Slick2D these screens are referred to as states.

If your game has only one state then there is no need to use a state-based game. Using a state-based game with only one state will lead to extra work that you can avoid by using a basic game. If your game only has one state and you plan to add other states later in development then you should plan ahead and use a state-based game.

Game states

A state is the equivalent of a basic game. Each state is an individual class and includes its own workflow consisting of an `init`, `update`, and `render` method. States do not necessarily need to include a constructor and there is no main method in any of the state classes.

Each state is referenced by its State ID which is defined inside the state's class in a method named `getID()`. This method looks as follows (where "0" is the state's id):

```
public int getID() {
  return 0;
}
```

The `getID()` method is written by the developer and it is important to ensure that none of your states share the same ID. Having states with the same ID will cause runtime errors.

Setting up a basic game state

Every state-based game needs at least one state. Follow these steps to create one:

1. Create and name the class and extend `BasicGameState`.
   ```
   public class OurFirstState extends BasicGameState {}
   ```
2. Add the `init` method and add a `throws` declaration for `SlickException`.
   ```
   public void init(GameContainer gc, StateBasedGame sbg) throws
   SlickException {}
   ```
3. Add the `render` method and add a `throws` declaration for `SlickException`.
   ```
   public void render(GameContainer gc, StateBasedGame sbg, Graphics
   g) throws SlickException {}
   ```
4. Add the `update` method and add a `throws` declaration for `SlickException`.
   ```
   public void update(GameContainer gc, StateBasedGame sbg,int DELTA)
   throws SlickException {}
   ```

Switching between states

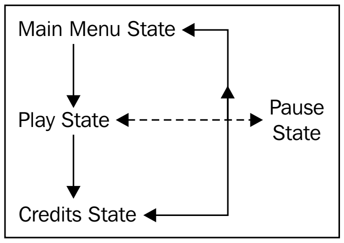
In a state-based game we switch between states using the `enterState(int ID)` method. We can switch states in any section of the workflow (`init`, `update`, `render`).

As you may have noticed from the previous section on *Setting up a basic game state*, all of the methods in the Slick2D workflow have a `StateBasedGame` parameter. This parameter only appears in state-based games and we can use it to switch between states. To switch between states we would simply call `enterState(int ID)` and use the ID we want to switch to for the parameter. For example, if we wanted to switch to a state with an ID of `2` we would write:

```
public void update(GameContainer gc, StateBasedGame sbg,int DELTA)
throws SlickException {
  sbg.enterState(2);
}
```

We can call `enterState()` from any of the three workflow methods along with the `enter()` and `leave()` functions that we will learn later in this chapter.

One way to help visualize the switching of states is to create state diagrams. This diagram will help you decide how the user is able to traverse between states. The following is an example of a state diagram:



Setting up a state-based game

We have already learned how to create game states, but how do we set up the window and start the game? When we set up a state, based game we need to consider the following:

- Creating a new class
- The constructor
- The main method

- The state list
- We perform the following steps:

1. First we need to create a new class that extends `StateBasedGame`.
2. The constructor should be created in the same manner as the constructor in the basic game. You should create the constructor in the following manner:

   ```
   public OurFirstStateBasedGame() {
     super("Our first state based game title");
   }
   ```

   The main method is set up the same way as in the basic game. We will learn about the main method in more depth later in this chapter.

3. For now we will only set the display mode (size of the window). A basic setup of the main method looks similar to the following code:

   ```
   public static void main(String[] args) {
     AppGameContainer agc = new AppGameContainer(new
   OurFirstStateBasedGame());
     agc.setDisplayMode(100,100,false);
     agc.start();
   }
   ```

   The state list is something exclusive to state-based games.

4. To create the state list we use a method called `initStatesList(GameContainer gc)`. The skeleton of this method looks similar to the following code:

   ```
     public void initStatesList(GameContainer gc) throws
   SlickException {
     }
   ```

   Don't forget to make a `throws` declaration or use a try-catch clause to handle slick exceptions.

   Inside the `initStatesList(GameContainer gc)` method we need to add all of the states that the game will use. We do this by using the `addState(GameState state)` method. The parameter for this method is an instance of the state.

5. Adding a state to the state list looks as follows:

   ```
   addState(new FirstState());
   ```

   Once we have added all the states to the state list we can enter a state. The last thing inside the `initStatesList(GameContainer gc)` method is the `enterState(int ID)` method.

6. Assuming we want to enter a state with a state ID of 0 it would look as follows:

   ```
   enterState(0);
   ```

Customizing state switching

Sometimes we need to execute some tasks when leaving and entering a state, such as saving values, resetting variables, and loading files. To do this we can override the `enter(GameContainer gc, StateBasedGame sbg)` and `leave(GameContainer gc, StateBasedGame sbg)` methods. These methods are overridden inside individual states and allow for optimal customization for the developer.

The `enter` method will be called right as the state is started. You can override it with anything you want. In the following example it prints a message to the console:

```
public void enter(GameContainer gc , StateBasedGame sbg) {
  System.out.println("Entering State!");
}
```

The `leave` method is the last thing called before the state is changed. Just like the `enter` method it can be overridden with anything you want. In the following example it prints a message to the console:

```
public void leave(GameContainer gc , StateBasedGame sbg) {
  System.out.println("Leaving State!");
}
```

Pre/Post state transition workflow

There are a few more features that state-based games have, involving the Slick2D workflow. These features include:

- Pre/Post state transition rendering
- Pre/Post state transition updating

These can be used in transitions, we will touch on this in *Chapter 7, From Example to Game*.

The app game container in all its greatness

The main method in Slick2D games includes a declaration of an app game container. We can customize this app game container in a multitude of ways. The only necessary customization is the display mode, but there are many other customizations available.

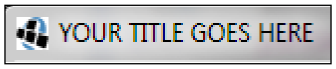
The Following is a collection of most of the customizations you can make to the app game container. It is important to remember that after all of these customizations, we must start the app game container using `start()`.

```
setFullScreen(boolean fullScreen);
```

If the parameter is true the window will be full screen and if it is false the window will be the size defined in the `setDisplayMode` method.

```
setIcon(String path);
```

The `setIcon` method sets the image found in the top right corner of the window.



```
setTitle(String title);
```

This method alters the title found in the top right corner of the window.

```
setMouseCusor(String path, int hotSpotX, int hotSpotY);
```

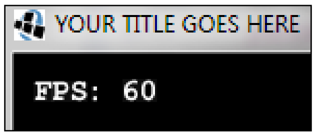
This method sets the image of the cursor inside the window. The first parameter is a path to the figure we want to act as the cursor. The second and third parameters are the x and y coordinates of the cursor hotspot. The hotspot is what pixel in the figure is used to determine the location of the cursor. For example, if the hotspot is at [0, 0] then the top left pixel of the figure will be used for the location of the cursor.

```
setUpdateOnlyWhenVisible(boolean updateOnlyWhenVisible);
```

This method decides whether or not the game will update if it is not visible on the user's screen.

```
setShowFPS(boolean show);
```

In Slick2D the fps is displayed in the top left corner. To disable this display use this method and pass `show` as `false`. If we pass `show` as true our screen will look as follows:



```
FPS: 60
```

FPS stands for **frames per second**. The number of frames per second is how many times the game runs the `render()` method. We will go over the render method in *Chapter 3*, *The Slick2D Workflow*, and in more depth in *Chapter 4*, *A Deeper Look at Rendering*.

```
setTargetFrameRate(int fps);
```

This method sets the maximum frame rate for the game. The game will automatically limit the frame rate to the amount set in this method.

```
setSoundOn(boolean on);
```

This method turns the sound on or off. Sound refers to sound effects not music.

```
setMusicOn(boolean on);
```

This method turns the music on or off.

We will look at the difference between sound and music, and how to implement it, in *Chapter 6*, *Sound and Music*.

Closing the window

To close the window we use the `destroy()` method. This will destroy the game container. This method should be called when quitting your game.

What we learned

Well we aren't off making Triple A titles quite yet, but we do have enough knowledge to get a window set up with all the bells and whistles. Throughout this chapter we covered an array of things including:

- The different game structures
- How to set up a basic game as well as a state-based game

- What should be included in the main method
- How to alter different options on the app game container

Starting our game

Throughout this book we will be creating a game using Slick2D. In each chapter, we will add a few things we learned to the game. If you want to see the source code for the finished game you can find it in *Appendix A*.

To start our game we will perform the following:

- Create a basic game named `Game`
- Add the constructor, workflow, and a main method
- Title the game
- Set up the app game container
    - The display mode should be 512 x 512 and not full screen
    - You may alter the game container in any other way you feel fit

If you cannot remember how to do any of the things mentioned previously, you can look back at the chapter; all of the things we added are from this chapter. If you still can't figure it out you can check out *Appendix A*.

Summary

We will continue to add things to this game throughout the book. I encourage you to not copy the code straight from *Appendix A* and instead look back at the chapter for help.

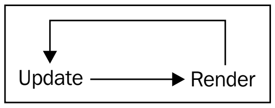
3
The Slick2D Workflow

In this chapter we will cover:

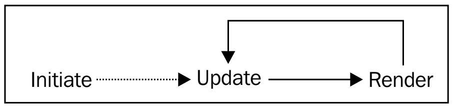
- Basic game workflow
- Initializing images
- Delta time / delta timers
- A look at rendering

Game workflow

A workflow is a sequence of events that transpires in order to get from initiation to completion. In game development the game's workflow is called the **game loop**. A typical game loop is update, render, and repeat. The Following is a diagram showing a basic game loop.



In Slick2D the workflow is almost identical. The only difference is the addition of the initiation event. Initiation is called only, once unlike update and render. It is used for loading images, setting variables, and any other tasks that we may require before we start the update and render loop. The Following is a diagram that shows the extended Slick2D game loop:



Using the workflow

In *Chapter 2, Game Structure*, we learned how to add the Slick2D workflow to both a basic game and game states. As long as we have these methods in either our basic game or our game state, depending on our game structure, Slick2D will automatically call these methods in a loop for us.

In Slick2D we do not have to set up a loop and call the functions that make up the workflow.

Remember that the initiation function will only be called upon entering a state or starting a basic game. If we leave a state and return to it later, the initiation function will be re-called.

The initiation method

The `initiation` method is a crucial aspect of the workflow that allows the developer to set up aspects of the game and initialize variables. It is called only one time during the first run of the game loop. It will be called only once and not repeated in the game loop and is called before both the update and the render method. The main uses for the `initiation` method include:

- Creating new instances of classes
- Initializing variables
- Loading images and other files
- Resource handling

Using the initiation method

The `initiation` method is written as `init()` in code. The name is shortened to `init` to allow for cleaner code. Let's take a deeper look at the `initiation` method.

```
init(GameContainer gc) {}
```

The `initiation` method has only one parameter: the game container. The game container is similar to the app game container we discussed in *Chapter 2, Game Structure*. We will go over the game container in depth later in this chapter.

In Slick2D it is important to include a `throws` declaration to all the aspects of the game workflow. In the `initiation` method this is to catch anything that was initiated incorrectly, such as images and files. The full declaration of the `initiation` method throws a `SlickException`.

```
public void init(GameContainer gc) throws SlickException {}
```

Alternatively we can use a try-catch inside each method within the game's workflow. This method may help you with debugging.

```
public void init(GameContainer gc) {
  try {

  } catch (SlickException e) {
  e.printStackTrace();
  }
}
```

The things we are initializing will go inside the try block.

Initializing images

In Slick2D one of the most important uses of the `initiation` method is loading images. We will go over images and how to render them in *Chapter 4, A deeper look at rendering*, but for now we will learn how to load them into the game.

Slick2D has its own image class separate from the image class located in pure Java. One mistake developers make when working with Slick2D is choosing the wrong image class. The Java image class is located in a package written as `java.awt.Image`. The image class we will be using with Slick2D is in a package written as `org.newdawn.slick.Image`.

> When creating a Slick2D image we must import the package `org.newdawn.slick.Image`.

To create a new Slick2D image we must perform the following tasks:

- Import the Slick2D image class
- Object declaration (creating an Image object)
- Creating a new instance of the image class

Let us go over the basic tasks required to create a new image in Slick2D and instantiate a new instance of it.

```
import org.newdawn.slick.Image;
```

The first step to creating an image in Slick2D is importing the correct image class.

[As with all imports in Java, you only have to import the class once and may use it an unlimited number of times.]

```
private Image image;
```

The next step is to declare a new Image object. In most cases, we will want our image object to be a private field. However if we plan to use the same image in multiple classes we should use the `public` keyword instead.

[The image object should be a field. In other words it should not be declared in a method, but instead within a class.]

```
public void init(GameContainer gc) throws SlickException {
  image = new Image("res/image.png");
}
```

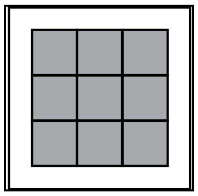
The final step is to create a new instance of an image. To do this we need to use the new keyword and pass a string to the constructor. The string we pass is the path to the image that we are trying to load. The path to our image in this case is inside the `res` folder. The location of your image may be wherever you like as long as the path you supply reaches that image.

[Ensure that you include the proper file type extension to your image path or the game will not be able to find the image at runtime.]

Loading images using a sprite sheet

If we create a game that has only a few images, the best way to load images is to use individual image files. However, if we were to make a larger game, we could use a more efficient method of loading images.

A sprite sheet is essentially one image that contains a selection of subimages referred to as sprites. The first step in using a Slick2D sprite sheet is creating the sprite sheet in an image editor. The Following is an example of a blank sprite sheet:



Each square in the sprite sheet represents one sprite. Through Slick2D we are then able to separate the sprites into individual images programmatically. Slick2D offers a very simple class that can be used to convert a sprite sheet into individual images. The steps to utilize this class include:

- Create a new sprite sheet
- Define the tile size
- Get the subimages

> The code used to use the sprite sheet should be inside the `initiation` method.

The first step in utilizing the sprite sheet class is to create a new `SpriteSheet` object and initialize it using one of the two constructors.

- If you have the sprite sheet saved in a Slick2D image already, you should use the following constructor:

  ```
  SpriteSheet(Image image, int tw, int th);
  ```

  This constructor takes the image containing the sprite sheet along with the width and height of the tile in pixels.

- If you have not yet loaded the sprite sheet into the game, you should use the following constructor:

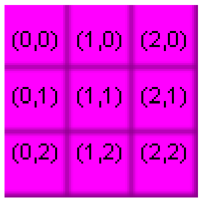
  ```
  SpriteSheet(String path, int tw, int th);
  ```

  This constructor takes the path to the image as a string, along with the width and height of the tile in pixels.

After we have initialized the sprite sheet and defined the tile size we can retrieve the individual subimages. Slick2D provides a very simple method to retrieve these images.

```
getSubImage(int x, int y);
```

The parameters for this method are the `x` and `y` values of the subimage you are trying to access. The x and y values are measured in tiles:



A basic setup for using the sprite sheet class is illustrated as follows.

```
SpriteSheet sheet = new SpriteSheet(""res/spritesheet.png"",16,16);
Image tile1 = sheet.getSubImage(0,0);
Image tile2 = sheet.getSubImage(1,0);
Image tile3 = sheet.getSubImage(0,1);
Image tile4 = sheet.getSubImage(1,1);
```

In our game we will only be using a few images so we will not be using a sprite sheet. If you are feeling ambitious, however, feel free to load images using a sprite sheet.

Converting buffered images to Slick2D images

While using Slick2D you may have to convert a buffered image into a Slick2D image.

In order to use many Slick methods, we need to use the Slick images as opposed to image classes packed into the **JDK** (**Java Development Kit**). Converting these files is best done in the `initiation` method, but may be done in any section of the workflow. To convert buffered images to Slick2D images perform the following steps:

```
public static Image toSlickImage(BufferedImage image) {}
```

The first step is to create a new method that we will use to convert images. In this case we will be creating a static method so that we can reuse the method in multiple classes. Of course, if you only need to use this method within one class, it may be best to remove the `static` keyword. Provide the method with a `BufferedImage` named `image` as a parameter and a return type of a Slick2D image.

```
Texture tex = null;
```

The first line within the method should be creating a new texture object and naming it. We will use this texture as a converter for the image.

```
try {
  tex = BufferedImageUtil.getTexture("HOLD", image);
} catch (IOException e) {
  e.printStackTrace();
}
```

The next few lines consist of a try-catch block. The try-catch block is to catch any input and output exceptions. Inside the try segment we will write one line creating a texture using the `BufferedImageUtil` and linking it to the texture we had created earlier. The first parameter for this method is an arbitrary string. You may supply any string of letters for this. The second parameter is a `BufferedImage`. We will pass the `BufferedImage` that we declared earlier. Inside the catch segment, we will simply print the stack trace.

```
return new Image(tex);
```

The final line in our method is the return statement. We will return a new `Image` instance using an `Image` constructor that takes a texture.

Our final method should look similar to to the following code:

```
public static Image toSlickImage(BufferedImage image) {
  Texture tex = null;
  try {
    tex = BufferedImageUtil.getTexture(""Blank Hold"", image);
  } catch (IOException e) {
    e.printStackTrace();
  }
  return new Image(tex);
}
```

The update method

The `update` method is a critical step in the Slick2D workflow. It should be used for changing values. It is a best practice to have no initialization or rendering done inside the update method, however, it should not cause problems. Unlike the `initialization` method the `update` method is called every time the game loop runs. The `update` method is called before the `render` method. The best uses of the `update` method include:

- Input handling
- Movement
- Value alteration
- Timers
- Collision detection

Using the update method

The `update` method is written `update()`. It is important to remember that all aspects of the Slick2D workflow require us to throw a `SlickException`. The full method is shown as follows:

```
public void update(GameContainer container, int delta) throws
SlickException{}
```

The parameters for the `update` method are an instance of the `GameContainer` and an integer named `delta`. We will go over the delta later in this chapter.

Alternatively we can use a try-catch inside each method within the game's workflow. This method may help you with debugging.

```
public void update(GameContainer gc, int delta) {
  try {

} catch (SlickException e) {
  e.printStackTrace();
  }
}
```

The things we are updating will go inside the try block.

Delta and how to utilize delta time

In the `update` method the second parameter is an integer named `delta`. This parameter is the amount of time in milliseconds that have passed since the last time that the `update` method was called by the game loop.

The **delta** is an invaluable tool that we can use to keep the game moving at the same speed no matter how many frames per second the game is running at. If we do not use the delta value, the game may run fine at 60 fps, but may be a completely different experience at 30 fps.

You can display the frames per second on the screen. We went over this in *Chapter 2, Game Structure*.

We can use the delta for many things including:

- Player movement
- Timers

Utilizing the delta variable in player movement

A great use for the `delta` variable is in player movement. We can use the delta variable to keep the game running at the same speed no matter what the frames per second is.

There are two main ways to integrate the delta with movement:

- Input delay
- Altering the movement speed

The more efficient way to integrate the delta is to use the delta to alter the movement speed. A basic example of how to alter the movement speed is shown as follows:

```
private int x,y;

public void update(GameContainer container, int delta) throws
SlickException {
  int movementSpeed = 1;

  x += (movementSpeed * delta);
  y += (movementSpeed * delta);
}
```

The way this example works is every millisecond that goes by the player will move by one pixel on both the x and y axis. because we multiplied the movement speed by the delta variable, the x and y axis will be increased by the delta. If we wanted the player to move twice as fast we could change the movement speed to 2.

[It is always a good idea to experiment with the movement speed to find one that makes the game run properly.]

Delta timers

A great use for the `delta` variable is a timer. Timers allow us to do many things within our game including animation and visual timers. Creating a timer with delta time is simple with Slick2D, because we can access the delta value straight from the `update` method. Let's go over how to create a delta timer.

```
Private int elapsedTime;
```

Create a new integer field and name it `elapsedTime`. This integer will keep track of how much time has passed.

```
private final int delay = 1000;
```

Create a new final integer field named `delay` and give it a value. This new integer field will instruct the program on how many milliseconds must elapse before the timer starts over. If we plan on altering the delay at any point we should remove the `final` keyword from the declaration.

[Remember that the delta variable is in milliseconds not seconds.]

```
if(elapsedTime >= delay) {
  elapsedTime = 0;
} else elapsedTime += delta;
```

Inside the `update` method create an `if` statement to test if the elapsed time has surpassed the delay. If the elapsed time has not surpassed the delay, then we will increase the elapsed time. Anything we want to alter when the timer goes off should be included in the `if` statement.

Input handling

Most input should be handled within the `update` method, however, you may include it in any part of the Slick2D workflow even though it is not a best practice. We will go over input handling in depth in *Chapter 5, A Look at Input*.

The render method

The `render` method is the powerhouse of the Slick2D workflow. It controls all of the graphics processing and drawing.

Slick2D makes rendering incredibly easy by eliminating many tedious tasks such as creating sharers or creating and using a sprite batch. Slick2D eliminates all of these tasks automatically so we don't have to. The things we will include inside of the `render` method include:

- Setting graphic options
- Drawing images
- Drawing graphics that are not images

Using the render method

It is important to remember that all aspects of the Slick2D workflow require us to throw a `SlickException`. The full method is shown as follows:

```
public void render(GameContainer container, Graphics g) throws
SlickException{}
```

Alternatively we can use a try-catch inside each method within the game's workflow. This method may help you with debugging.

```
public void render(GameContainer gc, Graphics g) {
  try {

  } catch (SlickException e) {
    e.printStackTrace();
  }
}
```

All of our rendering methods will go inside the try block.

Uses for rendering

Slick2D allows us to render a vast array of different things. The graphics parameter in the `render` method allows us to render a variety of things including:

- Lines
- Arcs
- Animations
- Oval/Circle
- Rectangle/Square
- Strings (of letters)

The Slick2D image class that was referred to earlier in this chapter has a method for drawing which allows us to draw the image without making a call to the graphics parameter. We may alternatively render images straight from the graphics object.

Rendering is for another day!

We will be going over rendering in much more detail in *Chapter 4, A Deeper Look at Rendering*. We will learn how to draw all the things listed previously, plus we will learn how to change attributes such as render color.

Workflow in a state-based game

In a state-based game the workflow is slightly different. One difference is a state-based game may have multiple workflows as each game state has its own workflow.

Another difference is that all the methods included in the workflow have an additional parameter. The parameter is an instance of the state-based game. The altered workflow is shown as follows:

```
init(GameContainer gc, StateBasedGame sbg) {}
render(GameContainer gc, StateBasedGame, Graphics g) {}
update(GameContainer gc, StateBasedGame sbg, int delta) {}
```

[Notice that the new parameter is not always added to the end of the parameter list.]

The new parameter is used to switch between states. We went over the state-based game parameter in depth in *Chapter 2, Game Structure*.

What we learned

We covered the entire Slick2D workflow and how we can utilize it in both a basic game structure and a state-based game structure. In this chapter we went over:

- How the Slick2D workflow works
- The `initiation` method
- The `update` method
- The `render` method
- Differences in workflow between the basic game structure and the state-based game structure

At this point, we have all the structures set up and we can now move onto actual gameplay aspects such as rendering and game mechanics.

Adding to our game

Throughout this book we are creating a game using the skills we learned from the current chapter. In each chapter we will add things to our game. We started the game in the last chapter and we will continue it here. If you want to see the source code for the finished game you can find it in *Appendix A*.

Let's add the following things to our game:

- Add the `initiation` method
- Add the `update` method
- Add the `render` method

We are also going to add a delta timer and load an image. I am going to help you along with these two additions, but if you are feeling ambitious, try to implement these features without any help.

Adding the delta timer

We will be using a delta timer inside the game. For now it has no function, but it will be a crucial aspect of our game. The first step is to set up a few variables. We need the following two fields:

```
Private int elapsedTime;
private final int DELAY = 1500;
```

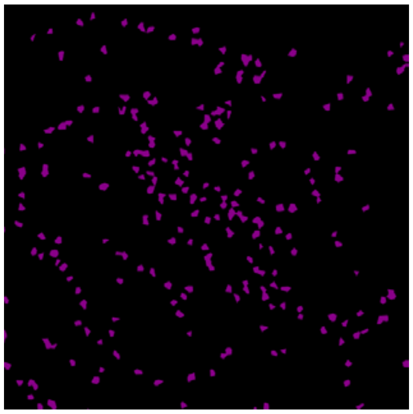
Next, we need to create the timer's `if` statement inside of the `update` method:

```
if(elapsedTime >= DELAY) {
  elapsedTime = 0;
} else elapsedTime++;
```

We will be adding the timer in future chapters.

Loading an image

We are going to add a background image to the game later in the book. For now, we are just going to load the image. The image you use may be different from mine but it should look somewhat similar to mine.



If nothing else, our image must be 512 x 512 pixels. After the image has been created we need to load it in. First we need to create an `Image` field.

```
private Image backGround;
```

All that's left now is to initialize it.

```
backGround = new Image("res/background.png");
```

If you cannot remember how to do any of the things above you can look back at the chapter; all of the things we added are from this chapter. If you still can't figure it out you can check out *Appendix A*.

Summary

We will continue to add things to this game throughout the book. I encourage you to not copy the code straight from *Appendix A* and instead look back at the chapter for help.

4
A Deeper Look at Rendering

Rendering

Rendering is an important part of computer graphics in which objects are drawn onto the screen. In Slick2D, we can draw an assortment of different things onto the screen from basic shapes to images.

[Rendering is often referred to as drawing.]

How Slick2D makes rendering easy

Slick2D provides an easy way for developers to render objects to the screen by abstracting the complex rendering pipeline of OpenGL.

Slick2D makes rendering easy for developers of all skill levels for this reason Slick2D is a great introduction to computer. Slick2D simplifies the following tasks required to render graphics to the screen:

- Vertex shaders
- Fragment shaders

In many other game and graphic libraries we would have to write our own shaders and create our own sprite batches.

What can we render?

Slick2D allows us to render both primitive shapes along with images. The primitive shapes include:

- Line
- Rectangle
- Rounded rectangle
- Oval/Circle
- Arcs

We can also render strings of characters to the screen using the `drawstring()` method.

Using the graphics parameter

In *Chapter 3, The Slick2D Workflow*, we went over the Slick2D workflow and the `render` method. As a quick review the `render` method is shown as follows:

```
public void render(GameContainer container, Graphics g) throws
SlickException{}
```

The only rendering we need to do in our game needs to be completed inside of this method. Many of the rendering tasks we need to accomplish can't be accomplished in other sections of the workflow, because the `render` method includes the graphics parameter.

The graphics parameter allows us to draw primitive shapes, strings, and images. Images can be drawn without directly calling the graphics parameter, we will go over image rendering in more depth, later in this chapter.

Drawing primitive shapes

There are several primitive shapes that Slick2D allows us to draw. A list of all the shapes can be found earlier in this chapter. All primitive shape rendering must be included in the `render` method.

Drawing a line

Lines are the simplest primitive that Slick2D allows us to draw. A line is simply a segment that starts from one location and ends at another. Unlike all the other primitives Slick2D offers, we cannot draw a filled version because it is not a shape.

Lines offer a wide variety of potential uses including:

- User interface
- Heads up display
- Borders
- Obstacles
- Player
- Enemy
- Terrain
- Segments in larger non-primitive shapes

To draw a line in Slick2D we use the `drawLine()` method. The full method is shown as follows:

```
g.drawLine(float x1, float y1, float x2, float y2);
```

The first two parameters are the x and y position of the start point of the line. The line primitive has no need for a bounding box.

When the line is only one pixel in size the start coordinates correspond to the first pixel of the line. However, if the line size is not one pixel then the parameters will correspond to the top pixel in the line.

The third and fourth parameters are the x and y coordinates of the end point of the line. Just as with the start point, when the line size is larger than one pixel the parameters will correspond to the top pixel in the line.

> The lines that we draw don't have to be completely straight. Slick2D allows us to draw shapes diagonally as well. To draw diagonal lines simply, ensure the start and end y values for our line are not equal.

Drawing rectangles

Drawing rectangles can be important for many tasks such as:

- User interface
- Heads up display
- Borders

To draw rectangles using the graphics parameter we need to call the `drawRect()` method. The full method is written as:

```
g.drawRect(float x1, float y1, float width, float height);
```

> The `g` before `drawRect` is the graphics parameter from the `render` method. We must include the `g` or we will get an error.

The first and second parameters are the x and y coordinates of the rectangle. The coordinate is for the top left corner of the rectangle assuming the width and height of our rectangle are positive.

The third and fourth parameter are the height and width of the rectangle.

> We can draw a square by setting the height and width of the rectangle to the same value.

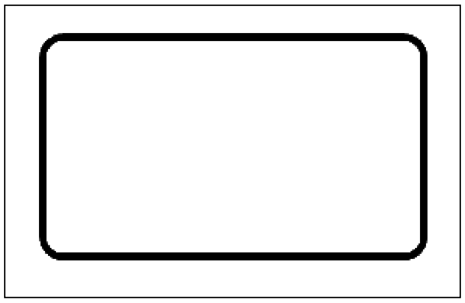
If we want to fill the rectangle in we can use the method `fillRect()`.

```
g.fillRect(float x1, float y1, float width, float height);
```

We will learn how to change the color of the fill later in this chapter.

Rounded rectangles

Slick2D allows us to draw a special type of rectangle called a rounded rectangle. This special rectangle has rounded corners giving it a more aesthetic look. An example of a rounded rectangle is displayed below.



To draw curved rectangles we use the `drawRoundRect()` method. The full method is shown as follows:

```
g.drawRoundRect(float x, float y, float width, float height,
intcornerRadius);
```

The first and second parameters are the position of the rectangle; these parameters are the same in both the rounded and regular rectangle.

The third and fourth parameters are the width and height of the rectangle; these parameters are same in both the rounded and regular rectangle.

The fifth parameter is the only different parameter. The parameter is how long the corner radius should be. A large integer will produce a more gradual round while a lower integer will be sharper.

The `cornerRadius` parameter affects all of the corners. Slick2D in its current state does not allow us to alter individual corners.

If we want to fill a rounded rectangle in we can use the `fillRoundRect()` method.

```
g.fillRoundRect(float x1, float y1, float width, float height,
intcornerRadius);
```

We will learn how to change the color of the fill later in this chapter.

Drawing ovals and circles

Ovals and circles can be used in a variety of ways in game development such as:

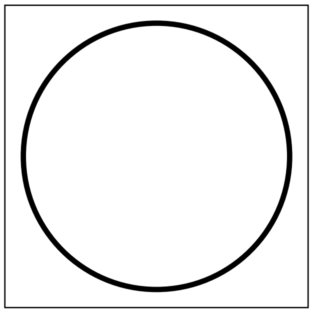
- Players
- Enemies
- User interface

To draw ovals and circles we use the `drawOval()` method. The full method is shown as follows:

```
g.drawOval(float x1, float y1, float width, float height);
```

The first and second parameters are the x and y coordinates of the oval we want to draw.

The oval creates a bounding rectangle to determine the position; the top left corner of this bounding rectangle is what the `x1` and `y1` parameters refer to. An example of an oval with its bounding rectangle is shown as follows:



The third and fourth parameters are the width and height of the oval we want to draw.

If we want to draw a perfect circle the width and height must be equal.

To draw filled ovals we use the `fillOval()` method.

```
g.fillOval(float x1, float y1, float width, float height);
```

We will go over how to change the fill color later in this chapter.

Alternative parameter list

The `drawOval()` and `fillOval()` methods also have an alternative parameter list to improve or worsen the segment quality.

An oval with bad segment quality will look more like a collection of line segments than an oval. To improve segment quality we need to specify that we want to draw the oval with a large amount of segments. The alternative methods for `drawOval()` and `fillOval()` are listed as follows:

```
g.drawOval(float x1, float y1, float width, float height, int
segments);
g.fillOval(float x1, float y1, float width, float height, int
segments);
```

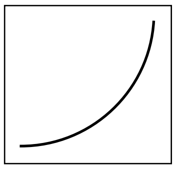
The only alteration to the original parameter list is the addition of the parameter segments.

To find a good balance for line segments we can simply try multiple amounts until we find one that looks appropriate. There is no perfect amount of line segments and tweaking is necessary due to the amount of variation we can have in the width and height of the oval. If we do not use the alternative parameter list Slick2D will default the segment count to fifty.

We can use very low segment counts such as five or six to create pentagons and hexagons.

Drawing Arcs

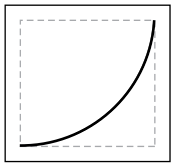
Arcs are similar to ovals except they don't have to be complete circles. Arcs start from one angle and end at another angle specified by the developer. A basic arc is shown in the following diagram:



To draw arcs we use the `drawArc()` method. The full method is shown as follows:.

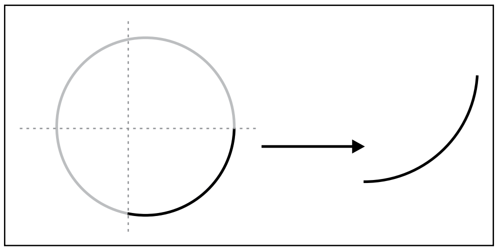
```
g.drawArc(float x1, float y1, float width, float height, float start,
float end);
```

The first and second parameters are for the x and y coordinates of the arc. The coordinates correspond to the top left corner of a bounding rectangle containing the entire arc. The bounding box is represented as:



The third and fourth parameters are for the height and width of the oval that the arc is sampled from.

All arcs are sampled from an arc. The following diagram demonstrates how an oval is sampled to form an arc:



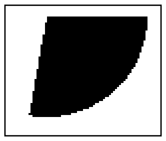
The fifth and sixth parameters are the start and end angles of the arc. The arc is sampled from an oval and the start and end angles mark the sample arc included inside the sampled oval.

[The start and end angle used in the fifth and sixth parameter should be written in degrees opposed to radians.]

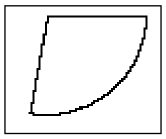
To draw filled arcs we use the `fillArc()` method.

```
g.fillArc(float x1, float y1, float width, float height, float start,
float end);
```

The arc does not enclose itself as in a circle or rectangle so filling an arc is not as straightforward. A filled arc in Slick2D is displayed below.



The way Slick2D handles filling arcs is by drawing a line segment from the ends of the arc to the center of the oval the arc is sampled from. Once these line segments are drawn we have a wedge, which we can then fill. The empty wedge is displayed as:



We will go over how to change the fill color later in this chapter.

Alternative parameter list

Much like the alternative parameter list referred to earlier in this chapter regarding ovals, the `drawArc()` and `fillArc()` methods both have an alternative parameter list to combat low segment quality.

The altered `drawArc()` and `fillArc()` methods are listed below.

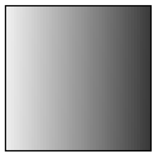
```
g.fillArc(float x1, float y1, float width, float height, int segments,
float start, float end);
g.drawArc(float x1, float y1, float width, float height, int segments,
float start, float end);
```

Just like with the altered methods in the ovals the only alteration in the parameter list is the addition of the parameter segments. However, notice that the new parameter is not added to the end of the current parameter list in both methods.

As with the alternative parameter list for the oval, the segments parameter will need to be tweaked to find the best fit. If we don't use the alternative parameter list, Slick2D will generate an arc using the smallest amount of segments possible while still keeping the shape.

Drawing gradient lines

Slick2D allows us to draw a special type of line called a gradient line. A gradient is when a color transitions from one color to another. An example of a gradient is shown as follows:



In the displayed gradient the color transitions from white to black by hitting several shades of grey along the way. Gradients are a successful way to slowly transition instead of jumping from one color to another.

Slick2D allows us to draw gradients along primitive lines that we went over earlier. An example of a line gradient is shown as follows:

The line gradient below goes from red to yellow. While creating a gradient we have two options:

Option 1 – Slick2D's color class

When creating a line gradient in Slick2d we use the `drawGradientLine()` method. The full method is displayed as follows:

```
g.drawGradientLine(float x1, float y1, Color Color1, float x2, float
y2, Color Color2);
```

The first two parameters are the `x` and `y` coordinates of the starting point of the gradient line.

The third parameter is a `Color` object corresponding to the starting color. To find a color we want, we need to access the color class in a static way. If we wanted the starting color to be red we would type `Color.red`. A full list of all the colors provided by the color class is shown as follows:

- Color.black
- Color.blue
- Color.cyan
- Color.darkGrey
- Color.grey
- Color.green
- Color.lightGrey
- Color.magenta
- Color.orange
- Color.pink
- Color.red
- Color.white
- Color.yellow

If we want to create our own color, we can, by passing along a new object. The constructor we want to use is listed as follows:

```
new Color(float r, float g, float b, float a);
```

The parameters are the red, green, blue, and alpha values of the color we want. If you plan on creating new colors it is best to use the second option for drawing gradient colors.

The fourth and fifth parameters are the x and y coordinates of the end position of the gradient line.

The sixth parameter is the end color of the gradient. We can consult the list of colors listed previously to find the color we want.

If we wanted to have a gradient that went from white to black using method one, the method would look similar to the following:

```
g.drawGradientLine(0,0,Color.white,100,0,Color.black);
```

Option 2 – define custom colors

In both options, we create a line gradient using the `drawGradientLine()` method. The method is altered in option two and is displayed as follows:.

```
g.drawGradientLine(float x1, float y1, float red1, float green1, float
blue1, float alpha1, float x2, float y2, float red2, float green2,
float blue2, float alpha2);
```

This method has a long list of parameters but is still very similar to the first option; the only difference being that we must define the color.

The first and second parameters are the x and y coordinates of the starting position of the gradient line.

The third, fourth, fifth, and sixth parameters are the values that make up the starting color of the gradient. If you are unsure what values go where, review how to make a new color in the first option to draw gradient lines.

[If we don't need to create an exact color it is best to use the first option to draw gradient lines and use some of the `Color` class's colors.]

The seventh and eighth parameters are the x and y coordinates of the ending position of the gradient line.

The ninth, tenth, eleventh, and twelfth parameters are the values that make up the ending color of the gradient.

If we want to have a gradient that goes from white to black using method two the method would look as follows:

```
g.drawGradientLine(0,0,1,1,1,1,100,0,0,0,0,1);
```

Drawing strings

Slick2D allows us to draw strings of characters to the screen by using the `drawString()` method. We can use strings for:

- User interface
- Titles
- Menus
- To draw a string all we have to do is call this method within the `render` method and access the graphics parameter.

```
g.drawString(String str, float x, float y);
```

The first parameter is the string we want to draw to the screen.

The second and third parameters are the x and y coordinates where the string should be drawn at.

Altering the behavior of the graphics object

The graphics parameter allows us to alter a variety of options that will alter the way objects draw. These options include:

- Anti-Aliasing
- Color
- Line width

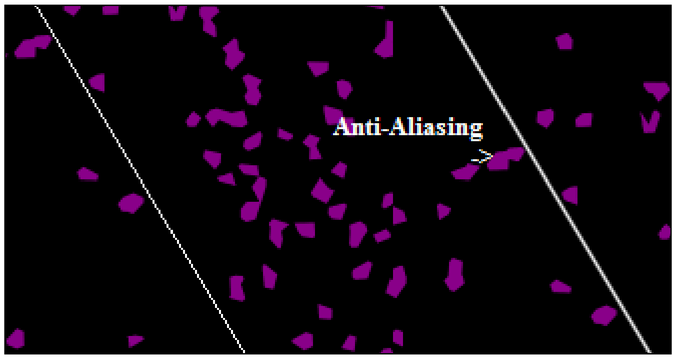
When we change these options they will stay altered unless we change the settings back during runtime.

Anti-Aliasing

Slick2D allows us the option to toggle anti-aliasing when drawing primitives. The method to toggle anti-aliasing is shown as follows:

```
g.setAntiAlias(boolean anti);
```

To toggle anti-aliasing we simply pass either `true` or `false` as the only parameter.



Color

Slick2D allows us to change the color that our primitives and strings are drawn with. The method to change the color is displayed below.

```
g.setColor(Colorcolor);
```

There is only one parameter for this method and it is the color that we want to set the graphics option to.

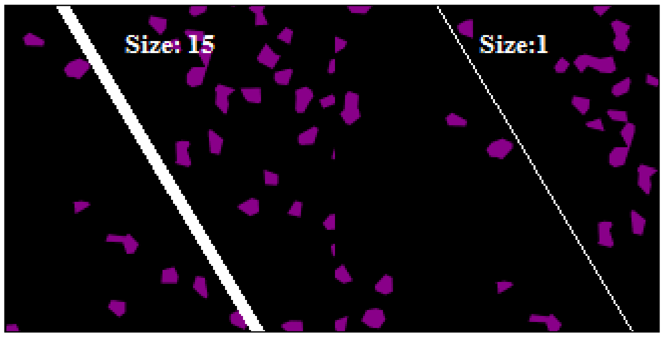
> The color will not revert back to the default color, we must call the `setColor()` method every time we want to change the color.

Line width

Line width is how many pixels thick the lines that make up primitives are. The method is shown as follows:

```
g.setLineWidth(float width);
```

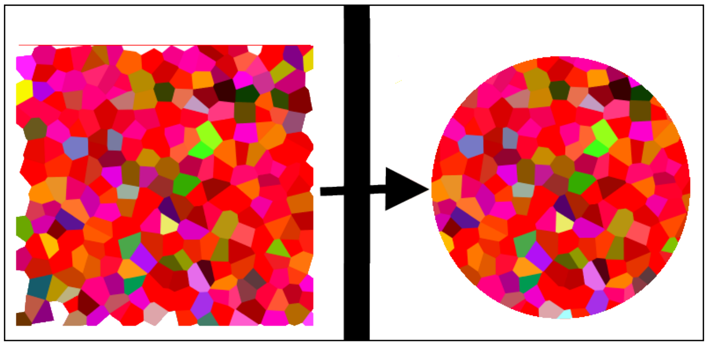
The only parameter for this method is the `width` we want to set for drawing lines in primitives.



Drawing textured shapes

Slick2D allows us to draw the primitive shapes referred to earlier in the chapter, with a texture. This allows us to draw an image in the shape of a primitive.

We will use the `texture()` method from the graphics library to accomplish this. The desired result is shown in the following diagram:



The method must be written inside of the `render` method. The code would look similar to the following:

```
public void render(GameContainer container, Graphics g) throws
SlickException{
  g.texture(new Circle(0,0,10),new Image("Path to our image"),
    1,1,true);
}
```

The first parameter is the shape we want to texture. In the previous example, we used a circle, but there are several shapes we could use including:

- Line
- Rectangle
- Curve
- Ellipse
- Circle

To find a list of all the shapes in Slick2D take a look at the javadocs located at `slick.ninjacave.com/javadoc`.

The second parameter is the image we will use to texture the primitive shape. We went over how to initialize images in *Chapter 3, The Slick2D Workflow*.

The third and fourth parameters are the scale for the x and y axis. If we don't want to scale it pass `1` for both parameters.

The fourth parameter is if we want to fit the texture to the shape. If true we want to fit it to the shape.

Drawing images

If we want to draw an image without texturing a shape we can use the `drawImage()` method. There are several different variations to the method allowing us to do things such as apply scale and color filters to our image. One of the more basic variations is displayed in the following code:

```
g.drawImage(Image image, float x, float y);
```

The first parameter is the image we want to draw. We went over how to create and initialize images in *Chapter 3, The Slick2D Workflow*.

The second and third parameters are the x and y values for where the image should be drawn.

Alternatively we can draw an image without directly calling the graphics parameter in the `render` method. This alternative way to draw images is more commonly used and it will be the way images are drawn in *Appendix A*. The appropriate way to draw using the image class is shown as follows:

```
image.draw(float x, float y);
```

The first and second parameters are the x and y coordinates where the image should be drawn.

[We must create the image object before we call the `draw` method. We learned how to create Slick2D images in *Chapter 3*, *The Slick2D Workflow* .]

Scaling images

Slick2D makes scaling images simple, all we have to do is tell Slick2D how much of a scale we want. We have two choices on how to scale images, either using the graphics parameter or using the image class. Drawing from the image class is preferred; the method we should use to scale the image is shown as follows:

```
image.draw(float x, float y, float scale);
```

The first and second parameters are the x and y coordinates where the image should be drawn.

The third parameter is the scale that should be applied to the image. If you don't want any scale, pass one to the third parameter.

[We don't have to draw the image and then scale it; scaling and drawing are both preformed in one method.]

If we want to shear the image we can use the `drawSheared()` method. This will draw the image slanted.

```
image.drawSheared(float x, float y, float hshear, float vshear);
```

The first and second parameters are the x and y coordinates where the image should be drawn.

The third and fourth parameters are how far the image should be sheared horizontally and vertically.

Shearing should not be confused with rotating an image which can be done with the `rotate()` method.

```
image.rotate(float angle);
```

The angle parameter is how far the image should be rotated in degrees. The image will be rotated around its center which we can define using the `setCenterOfRotation()` method.

```
Image.setCenterOfRotation(float x, float y);
```

The x and y values we pass as the center of rotation start from the top left hand corner of the image at (0,0).

Applying a color filter to an image

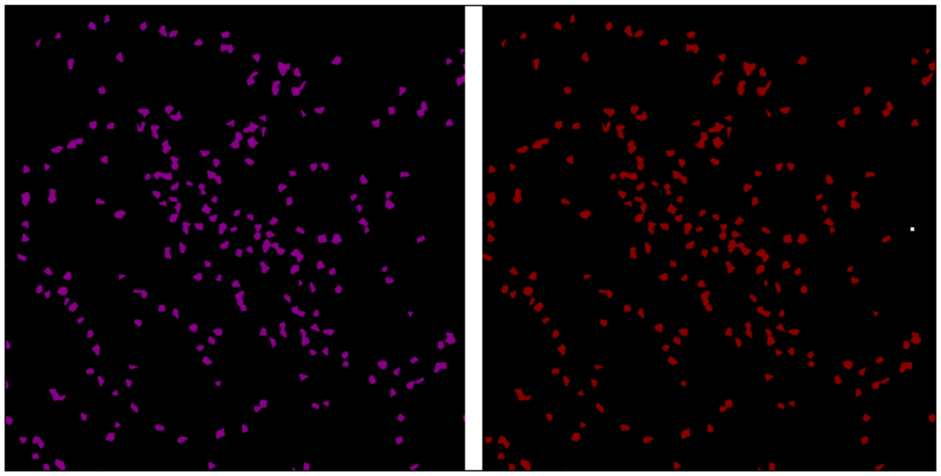
Slick2D allows us to apply a color filter to an image, which will alter the pixels in the image. The method is shown below.

```
image.draw(float x, float y, float scale, Color filter);
```

The first and second parameters are the x and y coordinates where the image should be drawn.

The third parameter is the scale referred to earlier in this chapter.

The fourth parameter is the color of the filter we want to apply. We went over the color class earlier in this chapter. If we apply a red color filter to an image it would have this effect.



What have we learned?

We hit a huge milestone in video game development, drawing something onto the screen. We learned how to draw an assortment of things to the screen including:

- Primitive shapes
- Lines
- Gradient lines
- Strings
- Images

We also learned how to alter the graphics options and how to apply different effects to images including scaling and color filtering. Now that we can get things onto the screen it's time to move them around.

Adding to our game

Throughout this book we are creating a game using the skills we learn from the current chapter. In each chapter we will add things to our game.

We added more to the game in the last chapter, including the game loop and an image. We will continue to add onto it here. If you want to see the source code for the finished game you can find it in *Appendix A*.

Let's add the following things to our game:

- Draw the image we loaded in the previous chapter
- Fill a square 5 x 5 pixels onto the screen
- Change the color of this square to red

If you cannot remember how to do any of the things mentioned previously you can look back at the chapter, all of the things we added are from this chapter. If you still can't figure it out you can check out *Appendix A*.

We will continue to add things to this game throughout the book. I encourage you to not copy the code straight from *Appendix A* and instead look back at the chapter for help.

Summary

If you are feeling ambitious, try adding a border around the screen as well using a line or even a gradient line. The code for this will not be included in *Appendix A* and should only be added if you think you can do it. You can check back through this chapter to review how to create lines and gradient lines.

5
A Look at Input

In this chapter we will cover:

- Uses for input
- Keyboard input handling
- Key codes
- Mouse input

The importance of user input

User input is what makes a game immersive and fun. A game without input isn't much of a game at all. User input allows the player to interact with the game instead of just watching the screen.

Uses of input

User input can be used for almost anything in your game. Game developers are always trying to find new and innovative ways to get the player involved in the game. A list of the more common uses for input are listed as follows:

- Player movement
- Camera movement
- Menu selection
- Actions (picking up items, opening doors, etc.)
- Attacking
- Puzzles

The uses for input are infinite and finding a new use for input could create a brand new game mechanic.

Using input for movement

One of the most common uses for user input is movement including player, camera, item, and so on. The most effective way to represent movement and position in two-dimensional space is using vectors. Vectors are used in both two-dimensional and three-dimensional games.

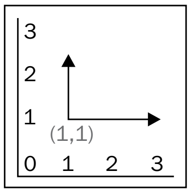
Basic overview of vectors

For the purpose of game development a vector has a position, direction, and magnitude. In Slick2D the most important aspect of vectors is the position. Let us break down a vector into its three components.

Vector position

Position is the location of the starting point of a vector. Vectors may be expressed in several dimensions. A one-dimensional vector will only have an x or y value, not both. A two-dimensional vector will have both an x and y value so that it may express its position in two dimensions. A three-dimensional vector will have an x, y, and z value to represent its value in three-dimensional space.

We will only be using two-dimensional vectors in Slick2D because Slick2D does not have support for three-dimensional games. A two-dimensional vector example is displayed in the following figure:



Vector direction

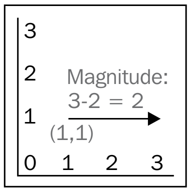
Each vector has a direction as well. The direction is which way the arrow points towards. Direction isn't very important in two-dimensional game development but still may be important during movement.

[The direction of each vector is independent of each other. Each vector has its own direction.]

Vector magnitude

The magnitude of a vector is the length of the arrow. The magnitude of a two-dimensional vector is not the combined length of both vectors.

[Slick2D refers to magnitude as length.]



Vectors in Slick2D

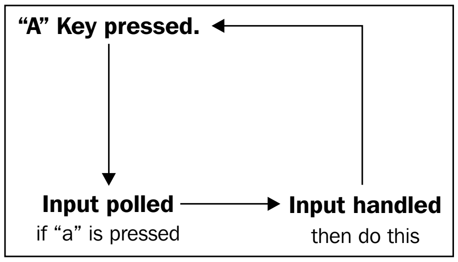
We will not go into vectors using Slick2D in depth because it is a complex topic, however, if you are feeling ambitious you could mess around with them. You can find the Slick2D javadocs that cover vectors at the following link:

`http://slick.ninjacave.com/javadoc/`

Vectors in Slick2D are included inside a class called `Vector2f`. The `2` is because the vector is two-dimensional. The `f` is because the position will be represented as a float variable.

Keyboard input

Slick2D handles keyboard input by polling the keyboard for input events. Input polling is a type of input handling that checks for input during each update. Unlike event-based input, the keyboard event will not trigger a method. This is explained in the following figure:



To handle keyboard input we first must create a new instance of the `input` class. The declaration is shown in the following code:

```
public void update(GameContainer gc, int delta)
   throws SlickException {
  Input input = gc.getInput();
}
```

> The input declaration must be done inside the `update` or `init` method and the `gc` is referencing the game container parameter.

To poll the input we have a few options. There are two main ways to poll the input, they are listed in the following sections.

Single press

The first way to poll input is finding a *single press*. A single press registers a key only once per press. In other words, if the user presses a key it will only process it once until you release and repress the key.

The method we need to use is: `isKeyPressed()` and it should be called from within the `update` method. The full method is displayed in the following line of code:

```
input.isKeyPressed(int code);
```

The only parameter is the key code that represents the key we want to poll. We will go over *key codes* shortly.

There are several uses for this type of input. The main uses are listed as follows, however, there are countless other uses.

- Typing
- Action button
- Toggling settings

Key down

The second way to poll input is polling input to find out if a specific key is pressed. The key down method allows us to test if a key is pressed every update without it having to be released first.

The method we need to use is: `isKeyDown()` and it should be called within the update method. The full method is displayed in the following line of code:

```
input.isKeyDown(int code);
```

Just like the `isKeyPressed()` the only parameter is the code for the key we want to poll. We will go over key codes shortly.

There are several uses for this type of input polling. Some common uses are listed as follows but there are infinite amount of possibilities to use it for.

- Movement
- Jumping

The most common use is movement because it would be frustrating for the user if they had to press and release the key every time to move.

Key codes

No matter what method we use to poll user input we always need to use key codes to poll specific keys.

Key codes are integers that represent a specific key on the keyboard. Luckily, Slick2D provides us with a set of final integer variables that we can reference from Slick2D's input class.

We don't have to memorize key codes for every key because of the `input` class, however, it is important to remember that the `input` class' key code variables represent an integer.

We need to access these key code variables from the input in a static way. In other words we need to access the class itself, not an instance of the class. An example of how to access the key codes is shown as follows:

```
Input.KEY_LEFT
Input.KEY_C
```

The basic syntax is the word `KEY` in all caps followed by an underscore and then the key we are looking for.

We can combine the methods of *user input polling* and *key codes* as displayed in the example given in the following code:

```
if(input.isKeyPressed(Input.KEY_LEFT) {}
if(input.isKeyDown(Input.KEY_LEFT) {}
```

A full list of all the key codes can be found in the javadocs currently hosted at:

`http://slick.ninjacave.com/javadoc/`

Using isKeyDown() to implement movement

Let's take a look at how we can use the `isKeyPressed()` method to move a rectangle around the screen.

The first step is to create two fields to keep track of the position of our rectangle. The variables should have the name `x` and `y`.

```
private int x;
private int y;
```

We will use these to decide where to draw the rectangle.

The next step is to set up the render method. We will need to use the render method to draw the rectangle.

```
public void render(GameContainer gc, Graphics g) {
  g.drawRect(x, y, 5, 5);
}
```

This will draw a rectangle 5 x 5 pixels in size at the location `x` and `y`. If we want to have the rectangle filled in we use `g.drawRect()`. the `render` method would then look like this.

```
public void render(GameContainer gc, Graphics g) {
  g.fillRect(x, y, 5, 5);
}
```

[If we were using a state-based game the render method would include the `StateBasedGame` parameter. The code to draw the rectangle would be unaffected.]

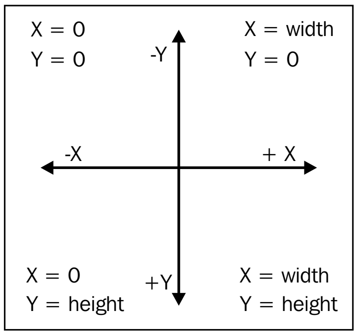
Next we will set up the `update` method. The `update` method will include all of our input polling. We will be using the arrow keys in this example.

```
public void update(GameContainer gc, int delta) {
  Input input = new gc.getInput();
  if(input.isKeyDown(Input.KEY_LEFT)) x-= 1 * delta;
  if(input.isKeyDown(Input.KEY_RIGHT)) x+= 1 * delta;
  if(input.isKeyDown(Input.KEY_UP)) y-= 1 * delta;
  if(input.isKeyDown(Input.KEY_DOWN))y+= 1 * delta;
}
```

We multiply one by delta so that the game runs and the rectangle moves at the same speed regardless of the frames per second.

Notice that when we press the up arrow key we are subtracting from the `y` variable and when we press the down arrow key we are adding to the `y` variable.

This may seem incorrect but remember that in Slick2D the pixel coordinates start in the top left corner of the screen.



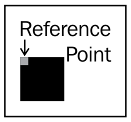
We can also add screen restraints to the movement so that our rectangle cannot move outside of the visible screen. If our screen is 512 x 512 the screen constraints would look like this.

```
if(x > 0) {
  if(input.isKeyDown(Input.KEY_LEFT)) x-= 1 * delta;
}
if(x < 507) {
  if(input.isKeyDown(Input.KEY_RIGHT)) x+= 1 * delta;
```

```
    }
    if(y > 0) {
      if(input.isKeyDown(Input.KEY_UP)) y-= 1 * delta;
    }
    if(y < 507) {
      if(input.isKeyDown(Input.KEY_DOWN))y+= 1 * delta;
    }
```

Now if you attempt to go off the screen your input will be ignored. This is a very basic form of collision detection.

We use `507` as our maximum screen constraint instead of 512 because the x and y coordinate of our rectangle is the top left corner. The following diagram draws the reference point in red.



To find maximum screen constraints, simply take the screen size subtracted by the movable's width or height.

Mouse input

While keyboard input is a crucial aspect of user input, mouse input allows us to create new game mechanics and make a game more fun. The mouse can be used for a collection of things including:

- User interface
- Menu
- Pause screen
- Movement

There are two main types of mouse input: mouse location and mouse click. Utilizing both in new and different ways can create fun new game mechanics.

Mouse location

Mouse location is the position of the mouse cursor within the game container. The input class keeps track of the mouse location.

We can access both the x and y coordinates of the mouse by using the `getMouseX()` and the `getMouseY()` methods located within the input method. These methods should be accessed through an instance of the input class. An example of these two methods is given in the following code:

```
public void update(GameContainer gc, int delta) {
  Input input = gc.getInput();
  input.getMouseX();
  input.getMouseY();
}
```

Mouse click

Polling for mouse clicks allows us to find out if the user has pressed the left, right, and middle button of the mouse. We will use the input class to poll the mouse for left, right, and middle clicks.

We access mouse clicks using either `mouseButtonDown()` or `mouseButtonClicked()`. Just like with keyboard input `mouseButtonDown()` polls the mouse to see if the mouse button is currently pressed. The `mouseButtonClicked()` polls the mouse to see if the mouse button has been released and then clicked again. Just like with polling the keyboard, mouse clicks should be processed in the update method.

The full method for `mouseButtonDown()` is displayed in the following line of code:

```
Input.mouseButtonDown(int KeyCode);
```

We use a key code to poll individual keys. We went over key codes earlier in this chapter. The main key codes for mouse clicks are listed as follows:

- Input. MOUSE_LEFT_BUTTON
- Input. MOUSE_RIGHT_BUTTON
- Input. MOUSE_MIDDLE_BUTTON

[Notice that these key codes do not start with the word key.]

The full method for `mouseButtonClicked()` is displayed as follows:

```
Input.mouseButtonClicked(int KeyCode);
```

The only parameter is the key code of the button we want to poll.

Combining mouse location and mouse click

One of the great things we can do with mouse input is combine the mouse location and the mouse click. One example of the use of both mouse location and mouse click is to draw a shape at the location of a click. The example is given in the following steps:

1. The first step is to create two fields for the x and y coordinates where we will draw an oval.

   ```
   public int x;
   public int y;
   ```

2. The next step is to set up the render method. Inside the render method we will be drawing the oval.

   ```
   public void render(GameContainer gc, Graphics g) {
     g.drawOval(x, y, 5, 5);
   }
   ```

3. We can also draw the oval filled in.

   ```
   public void render(GameContainer gc, Graphics g) {
     g.fillOval(x, y, 5, 5);
   }
   ```

4. The next step is to set up the update method. We will include both polling of the mouse location and the mouse click inside this method.

   ```
   public void update(GameContainer gc, int delta) {
     Input input = gc.getInput();
     if(input.mouseButtonClicked(Input. MOUSE_LEFT_BUTTON)) {
       x = input.getMouseX();
       y = input.getMouseY();
     }
   }
   ```

Don't forget to create the new instance of the `input` class or we will get a compile error.

The `update` method first tests to see if the mouse button has been clicked, and if it has, then we set the `x` and `y` coordinates to the mouse's coordinate.

If we didn't include the test for the mouse click the oval would simply follow the mouse at all times.

What we learned

In this chapter we learned how to bring our game to life. We learned how to utilize keyboard input using both `isKeyPressed()` and `isKeyDown()`. We learned what key codes are and what they represent. We also learned how to utilize the mouse's position and click to bring games to life.

User input is a crucial component of any game, input is what brings a game to life. Input allows us to create new and innovative game mechanics that make games fun and immersive.

Adding to our game

Throughout this book we are creating a game using the skills we learn from the current chapter. In each chapter we will add things to our game.

We added more to the game in the last chapter including the game loop and an image. We will continue to add onto it here. If you want to see the source code for the finished game you can find it in Appendix A.

Let's add the following things to the game:

- Use the arrow keys to move the rectangle we created in the last chapter
- Constrain the movement to the screen size

Summary

If you are feeling ambitious you could add a mouse position detector that prints the current x and y position to the screen. This will not be in the final game code in Appendix A, but would be good practice for using mouse input.

If you are feeling super ambitious you could try to replace the `x` and `y` variables we are using to keep track of the player's position with a Slick2D vector otherwise known as a `Vector2f`.

6
Sound and Music

In this chapter we will cover:

- The importance of sound and music
- Sound effect and music software
- Implementing sound and music into Slick2D games

The importance of sound and music

Sound and music plays a massive role in game development. Without sound and music games can be boring and dull. Games with repetitive sound effects and annoying music can be equally as bad. Sound effects and music are used in tons of aspects of game development including:

- Menu music
- Button click sounds
- Explosion sounds
- Attacking sounds
- Game music
- And much more…

Using music and sound as a game mechanic

While many games use sound and music to immerse the player in the game, some games use music and sound as a game mechanic.

A few games that use music and sound as a game mechanic are listed as follows:

- Guitar emulators
- Karaoke games
- Drumming simulations

Whether we use music and sound effects for immersion or as a game mechanic, music and sound effects are necessary for any game.

How to create sound effects and music

When it comes to using music and sound effects in Slick2D the hardest part is creating the music and sound. Implementing sound is easy thanks to Slick2D.

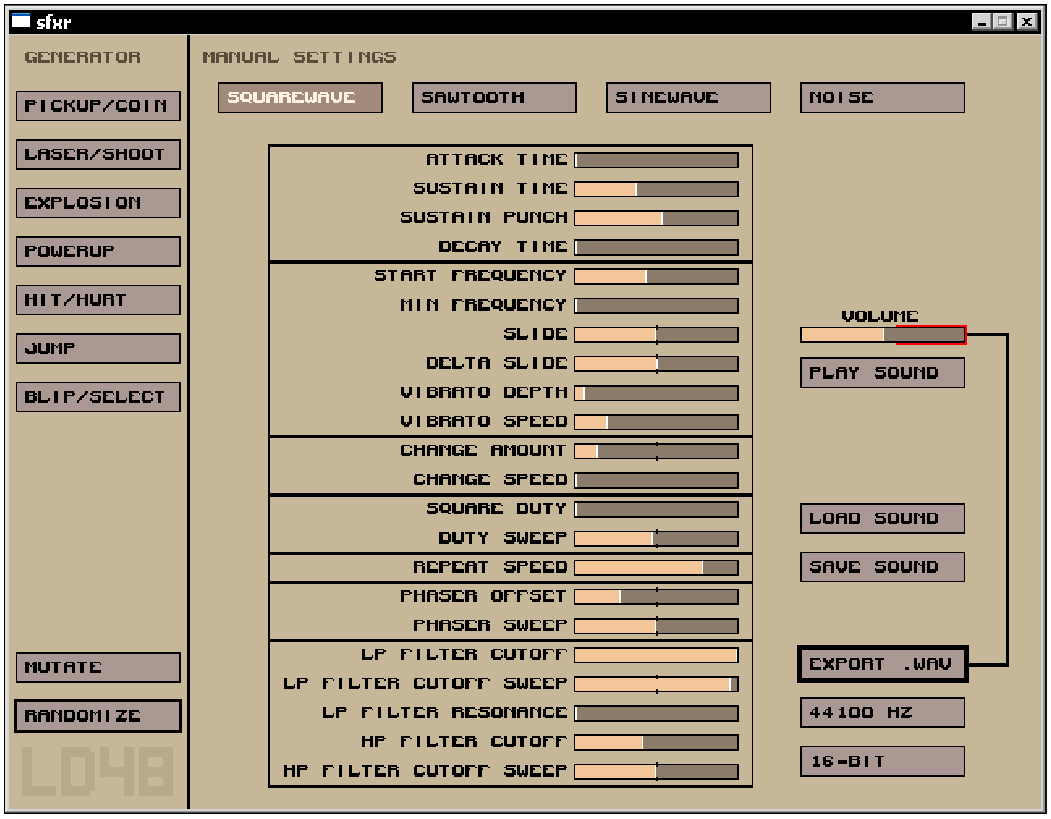
There are tons of music and sound effect creation software on the internet. We can find a couple of good ones for free.

Sound effect software

There are a few great sound effect creation tools on the internet for free. Some of the more common sound effect creation applications are listed as follows:

SFXR

SFXR is an easy to use sound effect generator that lets you create great sound effects without doing much work.

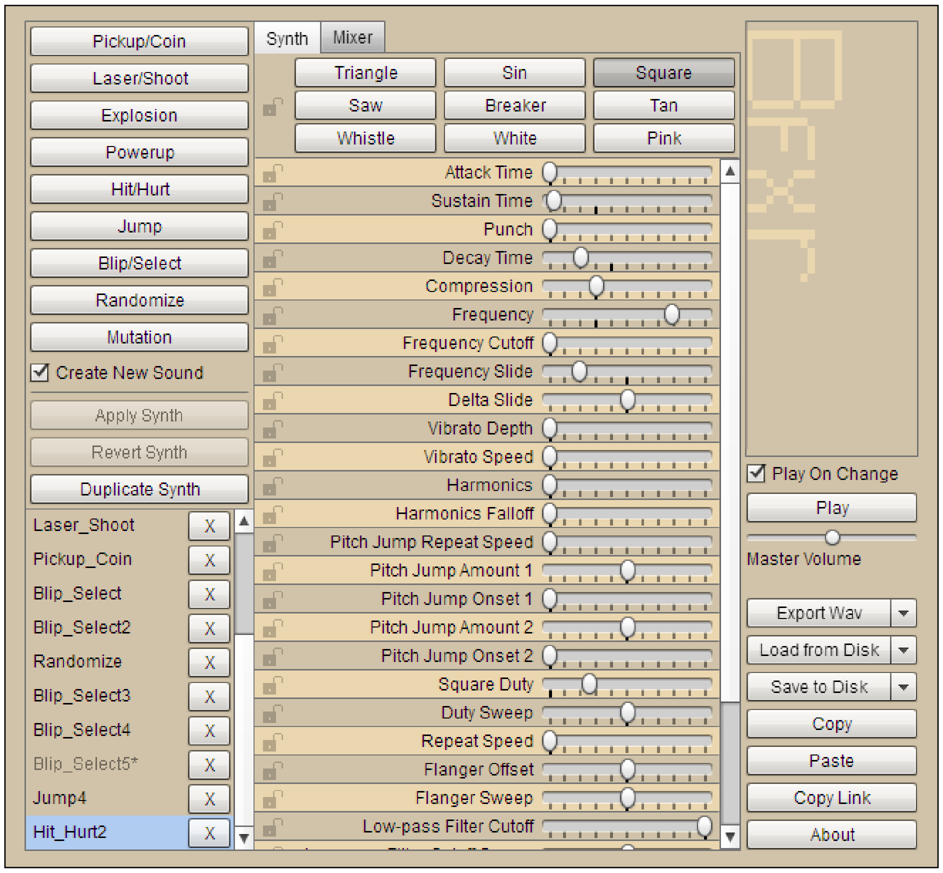


This application can downloaded by visiting its site, listed below:

`http://www.drpetter.se/project_sfxr.html`

BFXR

BFXR is a glorified version of SFXR including all the simplicity of SFXR but with more sound waves and more customization. BFXR has an online version along with a standalone application.



This application can downloaded by visiting the following site:

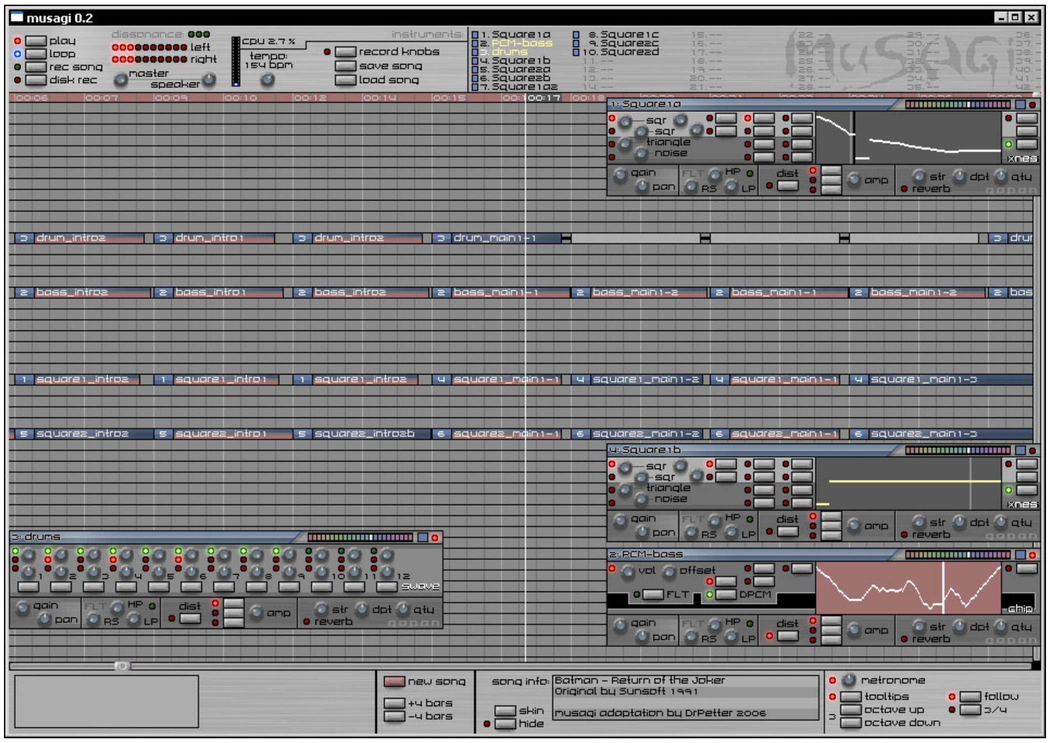
`http://www.bfxr.net/`

Music creation software

Music creation can be a daunting task, however, several applications make it a little bit easier.

Musagi

Musagi is a useful tool for music creation. It is a little complex and may be confusing for someone who doesn't have experience.



This music creation tool can be downloaded from the following link:

```
http://www.drpetter.se/project_musagi.html
```

Fruity Loops studio

Fruity Loops is a common choice for music creation. It has a large community so tutorials aren't hard to find.

The downside is this creation software is not free, however, the site does provide a free demo.

This music creation tool can be downloaded from the following link:

`http://www.image-line.com/documents/flstudio.html`

Using music in Slick2D

Once we have created some music or got some online all we have to do is add it into our game and play it. To add music to our game, we must first create a new instance of Slick2D's `music` class.

> Currently Slick2D only accepts music using the file type `OGG/MOD/XM`.

The only thing we need to provide to the constructor is the location of the music.

```
Music music = new Music("res/music.ogg");
```

Once the music is added we have two options for playing the music. We can either play the music normally or loop it so that it repeats.

To play we use the `play()` method.

```
music.play();
```

To loop the music we use the `loop()` method.

```
music.loop();
```

The `music` class allows us to play and loop music at a specific pitch and volume. The alternative parameter lists for these methods are shown as follows:

```
music.play(float pitch, float volume);
music.loop(float pitch, float volume);
```

The default value for pitch and volume is 1.0.

We have two options for stopping music, either pausing or stopping.

If we pause we can pick back up from where we stopped using the `resume()` method.

```
music.pause();
music.resume();
```

If we stop the music will not be able to be resume.

```
music.stop();
```

Music positioning

We can change the position of the music at any time. In other words we can decide what section of the song to play using the `setPosition()` method displayed below.

```
music.setPosition(float position);
```

The only parameter is the position in time we want to set. The time is measured in milliseconds.

We can also get the position at any time using the `getPosition()` method displayed as follows:

```
music.getPosition();
```

Fading music

We can fade music in and out using the `fade()` method. The full fade method is displayed as follows:

```
music.fade(int duration, float endVolume, boolean stopAfterFade);
```

The first parameter is the length of time in which the fade should take place. The time is measured in milliseconds.

The second parameter is the volume that the music should end at. Slick2D will slowly fade itself. this is the volume the music will end at.

The third parameter is a `Boolean` deciding if the music should be stopped after the fade is completed.

Altering and reviewing music variables

The music class allows us to alter the volume and access information about the current state of the music.

We can also change the volume of the music at any time using the `setVolume()` method displayed as follows:

```
music.setVolume(float volume);
```

We can also get the volume at any point using the `getVolume()` method displayed as follows:

```
music.getVolume();
```

If at any point we need to find out if the music is currently playing we can use the `playing()` method.

```
music.playing();
```

Using sound in Slick2D

Just like with music once the sound is created all we have to do is load it and play it. Music and sound are similar in Slick2D but we have less customization with sound with the absence of methods such as `setPostion()`.

[Slick2D refers to sound effects as just sounds.]

To create a new instance of Slick2D's `sound` class we only need the location of the sound effect.

```
Sound sound = new Sound("res/ourSound.ogg");
```

Just like music Slick2D currently supports `OGG/MOD/XM`.

Once we have loaded the sound into the game, using it is easy. We have two ways to play sound effects. Either `play()` or `loop()`.

```
sound.play();
```

Or:

```
sound.loop();
```

Just like the `music` class, the `sound` class allows us to alter the volume and pitch of the sound we want to play. The alternative parameter lists are displayed as follows:

```
sound.play(float pitch, float volume);
sound.loop(float pitch, float volume);
```

Unlike the `music` class there are no `pause()` and `resume()` methods. The only way to stop sound effects is wait till they finish playing or use the `stop()` method.

```
sound.stop();
```

At any time we can check if the current sound is playing by using the `playing()` method.

```
sound.playing();
```

Utilizing sound in an example

We have learned a lot about sound and music in this chapter so let's put everything we have learned together to make a soundboard.

We learned in previous chapters how to set up a basic game. To start out our soundboard let's create a new basic game with a screen size of 300 x 300.

With the game set up let's create two sound variables and initialize them in the `init()` method.

```
private Sound left, right;

//Inside init() method
left = new Sound("Path to sound");
right = new Sound("Path to sound");
```

Now, inside the update method, we need to gather input from the keyboard to test if the user has selected a sound. If the user has selected the left arrow key the program will play the left sound and if the user has selected the right arrow key the program will play the right sound.

```
//Inside the update() method
Input input = gc.getInput();

if(input.isKeyPressed(Input.KEY_LEFT)) {
  left.play();
} else if(input.isKeyPressed(Input.KEY_RIGHT)) {
  right.play();
}
```

Notice that when polling the input we use the method `isKeyPressed()` as opposed to `isKeyDown()` so that pressing the button does not spam the sound.

Now we have a working soundboard but it looks bland and only has two sounds. We can add more sounds by creating new `Sound` objects and utilizing more keys to play them. If we wanted to add a simple user interface to the soundboard we could create two rectangles and draw two strings indicating what the sound is. This is done in the following code:

```
//Inside the render() method
g.setColor(Color.red);
g.fillRect(0, 0, 150, 300);
g.setColor(Color.blue);
g.fillRect(150, 0, 150, 300);
g.setColor(Color.white);
g.drawString("Press the left \narrow for \nsound 1", 0, 150);
g.drawString("Press the right \narrow for \nsound 2" , 150, 150);
```

What have we learned?

In this chapter we learned how to utilize music and sound effects to give the player a more immersive experience. We learned the importance of music for immersion and as a game mechanic.

We also learned how to implement both music and sound effects in a game. We learned how to pause and resume music. We covered the differences between music and sound effects and the differences in options and flexibility they offer.

Adding to our game

Throughout this book we are creating a game using the skills that we learn from the current chapter. In each chapter we will add new features to our game.

We will continue to add onto it here. If you want to see the source code for the finished game you can find it in Appendix A.

In this chapter, adding to the book is completely optional. Music and sound effects can be difficult to create even with the programs referenced earlier in this chapter. This book isn't a guide to creating music and sound effects so if you would like to add them *great*, if not don't worry.

Summary

If you are up for making some music and sound effects try adding the following things to the game.

- Background music for the game that loops
- A sound effect that plays when you hit the screen boundaries that we discussed in the previous chapter.

These additions will make the game more immersive and *fun*, however they are not necessary. The additions we made this chapter are *not* included in the full source located in Appendix A.

7
From Example to Game

What we will cover

In this chapter we will cover the following:

- How to convert our example into a game
- Other features of Slick2D not covered in this book

The future of our game

Throughout this book we have added a few features to a game. In its current state the game isn't much fun, all we can do is move a square around the screen.

Well, it's about time that we turn our little example into a full game. The features we have so far included are as follows:

- A window
- Title
- Slick2D workflow
    - Initiation
    - Update
    - Render
- A delta timer
- An image
    - Loaded
    - Drawn to the screen

- A 5 x 5 red square keyboard input
- Screen constraints

We have had a good start so far but we need our game to have an objective and a reason to play.

Utilizing our delta timer

The first thing we are going to add to our game is the use of a delta timer. The use for our delta timer will be as a time constraint on the user.

The user's goal in our game will be to get to the specified side of the window before the timer runs out. The timer will be a major game mechanic in our game.

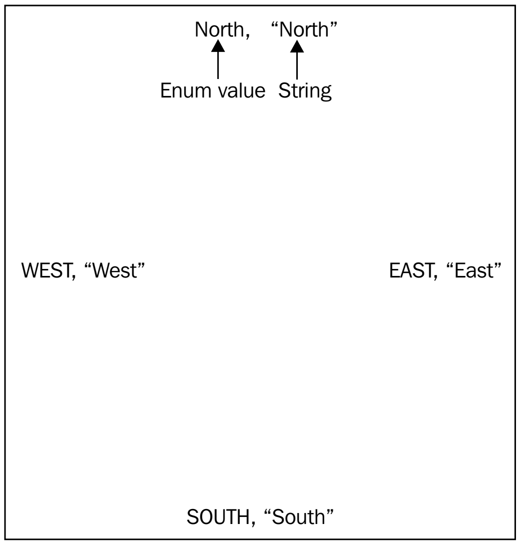
The delta timer needs to be altered slightly. The new delta timer is displayed in the following line of code, the additional *or-statement* is to ensure the player has not already gotten to the correct side.

```
if(elapsedTime >= DELAY || currentCardinal == currentLocation) {
}else elapsedTime += DELTA;
```

The first step is to define each of the sides of the window with cardinal directions.

Each of the sides will have a cardinal direction represented as a string and an `enum` value as shown in the following code:

```
private enum Cardinal {
  NORTH,
  SOUTH,
  WEST,
  EAST
}
private Cardinal currentCardinal = Cardinal.NORTH;
private String currentCardinalString = "North";
```



We will also add four `boolean` fields to track which side the player is touching as shown in the following line of code. We will go over how we will alter these later in the chapter.

```
private boolean isTouchN,isTouchS,isTouchW,isTouchE;
```

Before we proceed to add fields to our delta timer ensure that we have created the two fields `elapsedTime` and `DELAY` as shown in the following code:

```
private int elapsedTime;
private final int DELAY = 1500;
```

The delay I am using is 1500 milliseconds, however, you may alter this to balance the game.

Adding lives and points

Before we continue into the delta timer we need to add two more game mechanics to the game. These are as follows:

- Lives
- A point system

Lives

In our game we will give the player three lives. If the player does not reach the indicated side of the window before the delta timer ends, the player will lose a life.

> If you want you can alter this game mechanic, for example after enough points are earned the player may earn a new life.

To implement lives the first step is to create a new `integer` field named `lives`. The declaration is displayed in the following line of code:

```
private int lives = 3;
```

We will also need to create a `boolean` field to track if the player is alive or not. The declaration is displayed in the following line of code:

```
private bool alive = true;
```

We will go over the code necessary to implement lives later in the chapter when we finish the delta timer.

Point system

Our game will have a basic point system to give the player a new incentive to play. A point system also gives the player a sense of competition either against themselves or against a friend.

The first step to adding a point system to our game is creating an `integer` field to keep track of the current amount of points the player has. The code needed to create this declaration is given in the following line of code:

```
private int score;
```

The rest of the point system will be implemented inside of the delta timer. We will go over the delta timer later in this chapter.

Finishing the delta timer

Now it's time to bring the game together. Most of the game logic will be found inside the delta timer and it will handle the following things:

- Testing if we hit a wall
    - Setting new cardinal direction
    - Restarting delta timer
- Altering points
- Altering lives

The first step in finishing the delta timer is to create a new local variable inside the update method (not the delta timer) to keep track of the current direction.

> In `Cardinal currentLocation = null;` we will initialize the variable to null for now to ensure that it is altered by the timer.

Next we will create an `if-else` block to assign the `currentLocation` its actual value, this will be located in the update method outside of the delta timer.

```
if(isTouchN) currentLocation = Cardinal.NORTH;
else if(isTouchS) currentLocation = Cardinal.SOUTH;
else if(isTouchW) currentLocation = Cardinal.WEST;
else if(isTouchE) currentLocation = Cardinal.EAST;
```

The next step is to test if the player has touched the correct side. This will be located inside the delta timer.

```
if(currentLocation == currentCardinal && alive) points++;
else lives --;
```

We need to ensure that the player is still alive. This will also be located inside the delta timer.

```
if(lives < 0) alive = false;
```

After the user either doesn't get to the side in time or hits the correct side, the game will need to generate a new side to hit. To do this we will need to create a new instance of Java's `Random` class. This will be located out of all of the methods at the top next to the other fields as shown in the following code:

```
private Random rand;
```

To create the new instance we need to create a new `Random` class inside of the initiation method.

```
rand = new Random();
```

Next we will need to use the new instance to generate a new side. We will do this inside the delta timer. This is shown in the following code:

```
switch(rand.nextInt(4)) {
      case 0:
        currentCardinal = Cardinal.NORTH;
        currentCardinalString = "North";
        break;
      case 1:
        currentCardinal = Cardinal.SOUTH;
        currentCardinalString = "South";
        break;
      case 2:
        currentCardinal = Cardinal.WEST;
        currentCardinalString = "West";
        break;
      case 3:
        currentCardinal = Cardinal.EAST;
        currentCardinalString = "East";
        break;
      }
```

At this point the delta timer is finished. Ensure that we reset the timer. This can be done as shown in the following code:

```
elapsedTime = 0;
```

Checking for side collision

We need to check if the player has hit the side of the screen. In *Chapter 5, A Deeper Look at Rendering*, we added movement with screen constraints; these screen constraints will make side collision simple. To simplify screen constraints further let's create two final integer variables to account for the maximum and minimum screen positions. This is shown in the following code:

```
private final int screenMinimum = 0;
private final int screenMaximim = 507;
```

When we poll for input, the input detection should look as shown in the following code:

```
isTouchN=isTouchS=isTouchW=isTouchE=false;
if(x > screenMinimum) {
  if(input.isKeyDown(Input.KEY_LEFT)) x-= 1 * DELTA;
} else isTouchW = true;
```

```
if(x < screenMaximum) {
  if(input.isKeyDown(Input.KEY_RIGHT)) x+= 1 * DELTA;
} else isTouchE = true;
if(y > screenMinimum) {
  if(input.isKeyDown(Input.KEY_UP)) y-= 1 * DELTA;
} else isTouchN = true;
if(y < screenMaximum) {
  if(input.isKeyDown(Input.KEY_DOWN))y+= 1 * DELTA;
} else isTouchS = true;
```

We can simply test if we hit the screen constraint; if we didn't then we can continue to move in that direction. If we did then we have hit that side.

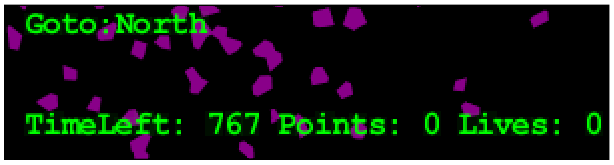
Adding visual feedback

With the current state of the game the user doesn't have any information. We need to give the user the following information:

- The cardinal direction to head towards
- Their lives
- Their points

The code to add a simple user interface that displays all this information is shown in the following code. It should be located inside the render method.

```
if(alive) {
  g.drawString("Goto:" + currentCardinalString, 200, 250);
  g.drawString("TimeLeft: " + (DELAY-elapsedTime) + " Points: "
     + points + " Lives: " + lives, 200 , 300);
  g.fillRect(x, y, 5, 5);
} else {
  g.setColor(Color.white);
  g.drawString("You Lose| Total Points: " + points, 200, 200);
}
```



```
Goto:North

TimeLeft: 767 Points: 0 Lives: 0
```

To add more depth to the game we will also switch the color of the player and the user interface depending on the current cardinal direction:

- Red = West
- White = East
- Orange = South
- Green = North

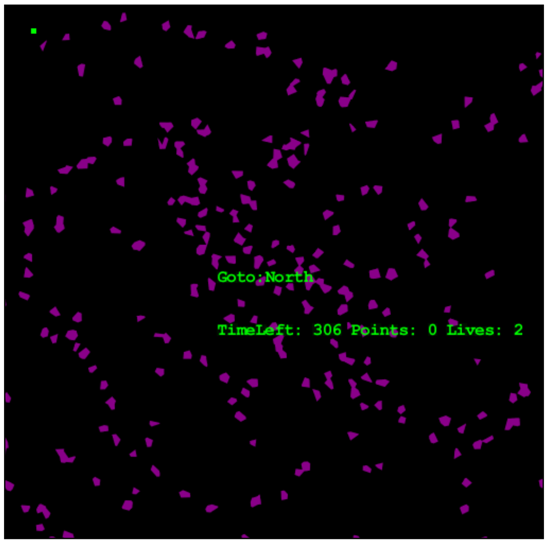
To add this visual feedback all we need to do is use the `setColor()` method. We also only want to alter the color if the player is alive. To implement color feedback simply add the following lines of code after inside the `if(alive)` block added previously as shown:

```
if(currentCardinal == Cardinal.NORTH) g.setColor(Color.green);
else if(currentCardinal == Cardinal.SOUTH)g.setColor(Color.orange);
else if(currentCardinal == Cardinal.WEST)g.setColor(Color.red);
else g.setColor(Color.white);
```

If you don't remember what the `setColor()` method means, refer to *Chapter 4, The Slick 2D workflow*.

Making the game unique

At this point the game is complete, we have a fun little game that demonstrates all of the skills we learned in the book.



If you want to take the game a little further you should add your own personal touch. If you want a more advanced game I recommend first going back and adding any features to the game that were listed under the *Feeling ambitious?* heading in previous chapters.

Some more suggestions include:

- Using a sprite sheet to load images (*Chapter 3*, *Game Structure*)
- Giving the player an image (*Chapter 3*, *Game Structure*, and *Chapter 4*, *The Slick 2D Workflow*)
- Changing music after each side hit (*Chapter 6*, *A Look at Input*)

Summary

In our final chapter we learned how to turn our boring old example game into a living game with an objective and reason.

Throughout this book we learned valuable information about the Slick2D library and game creation in general. We didn't hit everything Slick2D has to offer but we hit all the essentials. If you are feeling ambitious, try out some of the other Slick2D features including:

- XML reader
- Particle emitter
- Gradients
- Geometric shapes
- GUI
- Controller input

A
Full Source Code

The use of this Appendix

Throughout this book we worked on a small game to demonstrate the knowledge of all the skills we have learned. This appendix is simply a reference and should not be copied word-by-word. To really learn the concepts, follow the book and try to build the game along the way, if you are stuck the full source code is displayed below.

Full source code

```
import java.util.Random;

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.Color;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.Input;
import org.newdawn.slick.SlickException;

public class Game extends BasicGame {

  private int x = 25;
  private int y = 25;
  private boolean isTouchN,isTouchS,isTouchW,isTouchE;
  private Cardinal currentCardinal = Cardinal.NORTH;
     //N = 0 , S = 1 , W = 2 , E = 3
  private String currentCardinalString = "North";
```

```
public enum Cardinal {
  NORTH,
  SOUTH,
  WEST,
  EAST
}

private int points = 0;
private int lives = 3;
private boolean alive = true;

private int elapsedTime;
private final int DELAY = 1500;

private Image backGround;

private Random rand;

public Game() {
  super("Ongoing example");
}


public void render(GameContainer gc, Graphics g)
   throws SlickException {
  backGround.draw();
  if(alive) {
    if(currentCardinal == Cardinal.NORTH)
       g.setColor(Color.green);
      else if(currentCardinal == Cardinal.SOUTH)
        g.setColor(Color.orange);
    else if(currentCardinal == Cardinal.WEST)
      g.setColor(Color.red);
    else g.setColor(Color.white);

    g.drawString("Goto:" + currentCardinalString, 200, 250);
    g.drawString("TimeLeft: " + (DELAY-elapsedTime)
       + " Points: "  + points + " Lives: " + lives, 200 , 300);
    g.fillRect(x, y, 5, 5);
  } else {
    g.setColor(Color.white);
    g.drawString("You Lose| Total Points: " + points, 200, 200);
  }
}
```

```
public void init(GameContainer gc) throws SlickException {
  rand = new Random();
  backGround = new Image("res/background.png");
}


public void update(GameContainer gc, int delta)
   throws SlickException {
  Cardinal currentLocation = null;

  if(isTouchN) currentLocation = Cardinal.NORTH;
  else if(isTouchS) currentLocation = Cardinal.SOUTH;
  else if(isTouchW) currentLocation = Cardinal.WEST;
  else if(isTouchE) currentLocation = Cardinal.EAST;

  if(elapsedTime >= DELAY || currentCardinal == currentLocation)
     {
    if(currentLocation == currentCardinal && alive) points++;
    else lives --;

    if(lives < 0) alive = false;

    switch(rand.nextInt(4)) {
    case 0:
      currentCardinal = Cardinal.NORTH;
      currentCardinalString = "North";
      break;
    case 1:
      currentCardinal = Cardinal.SOUTH;
      currentCardinalString = "South";
      break;
    case 2:
      currentCardinal = Cardinal.WEST;
      currentCardinalString = "West";
      break;
    case 3:
      currentCardinal = Cardinal.EAST;
      currentCardinalString = "East";
      break;
    }

    elapsedTime = 0;
  } else elapsedTime += delta;
```

```
      Input input = gc.getInput();
      isTouchN=isTouchS=isTouchW=isTouchE=false;
      if(x > 0) {
        if(input.isKeyDown(Input.KEY_LEFT)) x-= 1 * delta;
      } else isTouchW = true;
      if(x < 507) {
        if(input.isKeyDown(Input.KEY_RIGHT)) x+= 1 * delta;
      } else isTouchE = true;
      if(y > 0) {
        if(input.isKeyDown(Input.KEY_UP)) y-= 1 * delta;
      } else isTouchN = true;
      if(y < 507) {
        if(input.isKeyDown(Input.KEY_DOWN))y+= 1 * delta;
      } else isTouchS = true;
    }

    public static void main(String args[]) throws SlickException {
      AppGameContainer agc = new AppGameContainer(new Game());
      agc.setDisplayMode( 512, 512, false);
      agc.setShowFPS(false);
      agc.start();
    }
  }
```

B
Packaging Our Game

Exporting a Slick2D game

Unfortunately, when we want to export our game for distribution, we have to create a *fat jar* which is an executable `JAR` file that includes the libraries used to create our game, including Slick2D and LWJGL.

Software

Luckily there is a software program called **JarSplice** that makes creating fat jars simple and easy. You can download JarSplice here for free: `http://ninjacave.com/jarsplice`.

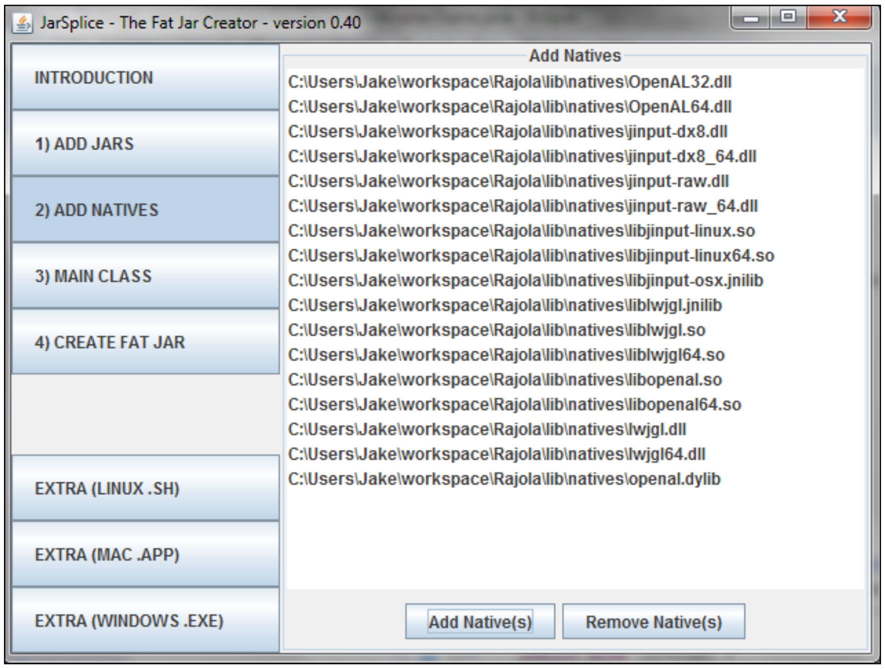
The site listed here has very nice documentation along with FAQs, so if you need more information check the site.

Packaging our game

Once we have downloaded Jarsplice it's time to create our fat jar file.

- The first step is to create a *non-runnable* jar file for our game.
  - Right-click on the game project and press **export**.
  - Export it as a `JAR` File. (Not a runnable `JAR`)
- Open up JarSplice

- Press the **Add Jars** tab
  - Add the JAR created in the first step
  - Add `slick.jar`
  - Add `lwjgl.jar`
- Press the **Add Natives** tab
  - Add the native files for all platforms you want to release on



```
C:\Users\Jake\workspace\Rajola\lib\natives\OpenAL32.dll
C:\Users\Jake\workspace\Rajola\lib\natives\OpenAL64.dll
C:\Users\Jake\workspace\Rajola\lib\natives\jinput-dx8.dll
C:\Users\Jake\workspace\Rajola\lib\natives\jinput-dx8_64.dll
C:\Users\Jake\workspace\Rajola\lib\natives\jinput-raw.dll
C:\Users\Jake\workspace\Rajola\lib\natives\jinput-raw_64.dll
C:\Users\Jake\workspace\Rajola\lib\natives\libjinput-linux.so
C:\Users\Jake\workspace\Rajola\lib\natives\libjinput-linux64.so
C:\Users\Jake\workspace\Rajola\lib\natives\libjinput-osx.jnilib
C:\Users\Jake\workspace\Rajola\lib\natives\liblwjgl.jnilib
C:\Users\Jake\workspace\Rajola\lib\natives\liblwjgl.so
C:\Users\Jake\workspace\Rajola\lib\natives\liblwjgl64.so
C:\Users\Jake\workspace\Rajola\lib\natives\libopenal.so
C:\Users\Jake\workspace\Rajola\lib\natives\libopenal64.so
C:\Users\Jake\workspace\Rajola\lib\natives\lwjgl.dll
C:\Users\Jake\workspace\Rajola\lib\natives\lwjgl64.dll
C:\Users\Jake\workspace\Rajola\lib\natives\openal.dylib
```

- Press the **Main Class** tab
  - Follow the on-screen directions to add your main class
- Press **Create Fat Jar**

Platform specific

JarSplice allows us to create platform specific executable files in addition to cross-platform JAR files.

To create these platform-specific files, just press the platform-specific tabs in the bottom left corner of the screen.

Index