Learning LibGDX Game Development
Second Edition

LibGDX is a multiplatform game development framework, and is one of the most popular frameworks for building games, providing a comprehensive set of development capabilities and features to use and incorporate into games. This book starts off by introducing you to LibGDX and how to use its project setup tool to build an application that can support multiple platforms, before gradually moving on to configuring your game. Then, you will learn to organize resources, create a scene, actors, and a menu system. You will also enhance the visual appearance of your game and manage its transitions, along with music and sound effects. You will then move on to cover the advanced programming techniques and create and manage different kinds of animations.

Finally, you will be introduced to the basics of 3D programming and the 3D physics engine. By the end of this book, you will be able to start developing your very own cross-platform games using the LibGDX framework.

Who this book is written for
This book is aimed at indie and existing game developers as well as those who want to get started with game development using LibGDX. Basic knowledge of Java programming and game development is required.

What you will learn from this book
- Set up a cross-platform project using Gradle and test the base code required for game building
- Speed up your overall productivity with the stunning JVM Code Hot Swapping feature
- Use Scene2D to create and organize complex menu structures
- Automate the creation of texture atlases using TexturePacker
- Manage and play audio files and add special effects to your game to improve its look and feel
- Learn 2D physics simulation using Box2D
- Understand more about 3D programming using the new LibGDX 3D API
- Use the LibGDX Bullet wrapper for 3D physics simulation
- Learn 2D physics simulation using Box2D
- Understand more about 3D programming using the new LibGDX 3D API
- Use the LibGDX Bullet wrapper for 3D physics simulation

In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 2 "Cross-platform Development – Build Once, Deploy Anywhere"
- A synopsis of the book’s content
- More information on Learning LibGDX Game Development Second Edition

About the Authors

Suryakumar Balakrishnan Nair is an engineering graduate from Cochin University of Science and Technology, Cochin, India with a specialization in computer science. He just loves programming and likes to keep on experimenting. He has designed a dozen games on the Android platform using LibGDX.

He loves traveling and visiting various places. He reads articles and books on a range of issues from politics to environment. He is currently working as a full-time Android game developer for an Indian game company, Csharks (http://csharks.com/site/).

I would like to thank my colleagues in Csharks for providing me with moral support, especially Vipin TP and Dheeraj S. I would also like to thank my dear friend Rahul Satish who helped me with the Blender models. Most importantly, I want to thank my mentor, Juwal Bose, who guided me and motivated me for this project.

Andreas Oehlke is a professional software engineer and computer scientist who feels very much at home on any Linux/UNIX machine. He holds a bachelor's degree in Computer Science and loves to assemble and disassemble software and hardware alike. The exorbitant affinity for electronics and computers has always been his trademark. His hobbies include game and web development, programming in general (software design and new languages), programming embedded systems with microcontrollers, playing sports, and making music.

He currently works full time as a software engineer for a German financial institution. Furthermore, he has worked as a consultant and game developer in San Francisco, CA.
his spare time, he provides his expertise to a German start-up called Gamerald (http://www.gamerald.com/).

I want to thank my parents, Michael and Sigrid, and my brother Dennis for their constant and invaluable support, which definitely kept me on the go while writing this book. I also want to thank my close friends for giving me lots of helpful feedback, notably Sascha Björn Bolz for providing the artwork for Canyon Bunny. Last but not least, I want to thank Klaus "keith303" Spang for providing the music track, the whole team of Packt Publishing, and all the numerous reviewers for their great work who helped me produce a high-quality book.
As personal computers have conquered our private homes, video games have become more and more popular and eventually a multimillion dollar business for big video game companies. With the introduction of mobile devices such as smartphones and tablets, the market for video games has experienced another significant increase; in particular, it has now become open to independent game developers with small budgets.

For game developers, it is essential to have tools at hand that provide fundamentals that allow rapid prototyping and cost-effective implementation of their creative ideas. This is where LibGDX comes into play. LibGDX, as a Java-based game development framework, provides a unified access layer to handle all the supported platforms. LibGDX also makes use of C/C++ to achieve cross-platform support as well as to boost the application performance for mission critical tasks.

This book will show you how easy it is to develop cross-platform games by walking you through a complete game development cycle using the free and open source library—LibGDX. Besides this, you will also learn about common game structure and the involved requirements.

You will be introduced to the key features of LibGDX. You will also learn how to develop a game with ease and speed up your development cycles. In ten easy-to-follow chapters, you will develop your first LibGDX cross-platform game and add more and more game functionalities as you progress further through this book.

The special features will also make you acquainted with advanced programming techniques such as animations, physics simulations, and shader programs that enhance your games in both their gameplay and visual presentation.

By the end of this book, you will have a fully working 2D game that will run on Windows, Linux, Mac OS X, WebGL-capable browsers, Android, and iOS. You will also have all the skills required to extend the game further or to start developing your own cross-platform games.

What This Book Covers

Chapter 1, Introduction to LibGDX and Project Setup, covers how to install and configure the development environment and introduces you to the project setup tool that comes with LibGDX. Then, we will take a first look at the basics of what a game needs to come alive.
Chapter 2, Cross-platform Development – Build Once, Deploy Anywhere, explains the supported target platforms and how to deploy and run our application on each platform using a demo application. For the first overview of LibGDX's API, we will take a glance at each module. Then, the application cycle will be introduced, and we will take a look at how to debug and manipulate our code at runtime.

Chapter 3, Configuring the Game, takes us from our demo application to a real game by setting up a new project called Canyon Bunny. We will work on this project throughout the rest of the book and extend it from chapter to chapter with new features. As LibGDX is a framework, we will first have to build our program architecture using UML class diagrams to structure our game engine.

Chapter 4, Gathering Resources, describes how to gather all the resources (assets) needed for Canyon Bunny, including graphics, audio files, level data, and so on. We will also find out how to load, track, and organize assets efficiently. Finally, it is time to think about how level data is going to be handled so that we are able to populate our game world with objects.

Chapter 5, Making a Scene, will implement the game objects such as rocks, mountains, and clouds. We will put the new code into action using a level loader. We will also add a Graphical User Interface (GUI) to the game scene to show the player's score, extra lives and frames per second to measure the games performance.

Chapter 6, Adding the Actors, explains how to add the remaining game objects for Canyon Bunny, including the player character and collectible items to complete our game. We will also add simple physics for player movement and basic collision detection. Additionally, the game logic will be extended so that it is able to detect the "life lost" and "game over" conditions.

Chapter 7, Menus and Options, describes how to create a menu system with widgets such as buttons, labels, and checkboxes to enrich the overall game experience. Furthermore, we will add an Options window where the player can adjust the game settings.

Chapter 8, Special Effects, covers how to make use of particle systems and how to apply interpolation algorithms to create impressive effects such as dust clouds, a smooth, following camera, floating rocks, and parallax scrolling for mountains in the background. Using special effects will spice up the appearance of your game.

Chapter 9, Screen Transitions, introduces screen transitions. We will dive into enhanced visual effects using OpenGL's Framebuffer Objects for off-screen rendering into video memory. This will allow us to create seamless transitions for an improved user experience while switching from one screen to another. For Canyon Bunny, we will create a variety of transition effects.

Chapter 10, Managing the Music and Sound Effects, will walk you through a list of recommended sound generators and discuss their differences. Then, we will take a look at...
the LibGDX's Audio API and demonstrate how to use it by creating an audio manager. We do this so that handling our entire audio playback needs become a breeze.

Chapter 11, Advanced Programming Techniques, introduces you to some advanced programming techniques that will guide you to the next level of game programming. We will build basic knowledge about the Box2D API that enables us to create believable physics simulations in games. Additionally, we will discuss the topic of shader programs with the example of a monochrome image filter effect. Lastly, we will show you how to make use of the accelerometer hardware that is commonly available in modern smartphones and tablets, which allows controlling the game by tilting the device.

Chapter 12, Animations, explains how to polish the game by adding animations. In this chapter, we will cover two different approaches to animate the game menu and the game world. Finally, we will implement a state machine to allow event-based animations for the player character.

Chapter 13, Basic 3D Programming, introduces the new LibGDX's 3D API. You will learn how to use the 3D API to create basic models such as sphere, cube, cylinder, and so on, and load models exported from modeling software such as Blender. You will also learn about ray picking, an important concept used to develop first person shooter games.

Chapter 14, Bullet Physics, will walk you through the basics of 3D physics using Bullet. Finally, we will create a simple application to simulate physics using Bullet.
In this chapter, you will learn more about the generated Eclipse projects and how they work together. Also, you will learn the following components of the LibGDX framework:

- Backends
- Modules
- Application life cycle and interface
- Starter classes

At the end of this chapter, you will take a closer look at the demo application and inspect the generated code of the Main class in great detail. You will learn how to set breakpoints, run the application in the debug mode, and speed up your overall productivity with the awesome JVM Code Hot Swapping feature. The discussion on the demo application ends with some simple and fun modifications to the code accompanied by a demonstration of the JVM Code Hot Swapping feature.

After completing this chapter, you will be able to deploy, run, and debug the demo application from Chapter 1, Introduction to LibGDX and Project Setup, on a desktop (including Windows, Linux, and Mac OS X), on Android, iOS, and in a WebGL-capable web browser such as Google Chrome.
The demo application – how the projects work together

In Chapter 1, Introduction to LibGDX and Project Setup, we successfully created our demo application, but we did not look at how all the Eclipse projects work together. Take a look at the following figure to understand and familiarize yourself with the configuration pattern that all your LibGDX applications will have in common:

What you see here is a compact view of four projects. The demo project to the very left contains the shared code that is referenced (added to the build path) by all other platform-specific projects. The main class of the demo application is MyDemo.java. However, there is a different main class where an application gets started by the operating system, which will be referred to as starter classes from now on. Notice that LibGDX uses the term starter class to distinguish between these two types of main classes in order to avoid confusion. We will cover everything related to the topic of starter classes later.

While taking a closer look at all these directories in the preceding figure, you might have spotted that there are two assets folders: one in the demo-desktop project and another in the demo-android project. This brings us to the question, where should we put all the application’s assets? The demo-android project plays a special role in this case. In the preceding screenshot, you can see a subfolder called data, which contains an image named libgdx.png. This image also appears in the demo-desktop project in the same place.
Just remember to always put all your assets into the assets folder under the demo-android project. The reason behind this is that the Android build process requires direct access to the application's assets folder. During its build process, a Java source file, R.java, will be automatically generated under the gen folder. It contains special information for Android about the available assets. It will be the usual way to access assets through the Java code if you were explicitly writing an Android application. However, in LibGDX, you will want to stay independent of the platform as much as possible and access any resource such as assets only through the methods provided by LibGDX. You will learn more about accessing resources in the last section of this chapter.

You might wonder how other platform-specific projects will be able to access the very same assets without having to maintain several copies per project. Needless to say this would require you to keep all copies manually synchronized each time the assets change.

Luckily, this problem has already been taken care of by the generator. The demo-desktop project uses a linked resource—a feature by Eclipse—to add existing files or folders to other places in a workspace. You can check this out by right-clicking on the demo-desktop project, navigating to Properties | Resource | Linked Resources, and then clicking on the Linked Resources tab.

The demo-html project requires another approach as Google Web Toolkit (GWT) has a different build process compared to other projects. There is a special file called GwtDefinition.gwt.xml that allows you to set the asset path by setting the gdx.assetpath configuration property to the assets folder of the Android project. Notice that it is good practice to use relative paths such as ../android/assets so that the reference does not get broken if the workspace is moved from its original location. Take this advice as a precaution to protect you and your fellow developers from wasting precious time on something that can be easily avoided by using the right setup, right from the beginning.

The following is the code listing for GwtDefinition.gwt.xml from demo-html:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<module>
```
Cross-platform Development – Build Once, Deploy Anywhere

Similar to the demo-html project, the demo-robovm project has a special file called robovm.xml that saves the path to the assets folder in demo-android. Notice the <directory> key under <resources>, where the relative path to the assets folder is set. However, this is not the end of resource setting for demo-robovm. In iOS projects, there will be some resources specific to iOS, such as icons and default splash images. You don't want to put this in your Android assets folder. So, put this in the folder named data in your demo-robovm project. The path of the folder is also linked in the robovm.xml file under <resources>.

Unlike Android, iOS version needs specific names for icons to show in respective devices. For example, Icon-72.png is the name for the app icon on iPad. You can find specifics of the icon name and size at https://developer.apple.com/library/iOs/qa/qa1686/_index.html.

The following code snippet is taken from robovm.xml in our demo-robovm project:

```xml
<config>
    <executableName>${app.executable}</executableName>
    <mainClass>${app.mainclass}</mainClass>
    <os>ios</os>
    <arch>thumbv7</arch>
    <target>ios</target>
    <iosInfoPList>Info.plist.xml</iosInfoPList>
    <resources>
        <resource>
            <directory>../android/assets</directory>
            <includes>
                <include>**</include>
            </includes>
            <skipPngCrush>true</skipPngCrush>
        </resource>
    </resources>
</config>
```
LibGDX backends

LibGDX makes use of several other libraries to interface the specifics of each platform in order to provide cross-platform support for your applications. Generally, a backend is what enables LibgGDX to access the corresponding platform functionalities when one of the abstracted (platform-independent) LibGDX methods is called; for example, drawing an image in the upper-left corner of the screen, playing a sound file at a volume of 80 percent, or reading and writing from/to a file.

LibGDX currently provides the following four backends:

- Lightweight Java Game Library (LWJGL)
- Android
- JavaScript/WebGL
- iOS/RoboVM
Lightweight Java Game Library

Lightweight Java Game Library (LWJGL) is an open source Java library originally started by Caspian Rychlik-Prince to ease game development in terms of accessing the hardware resources on desktop systems. In LibGDX, LWJGL is used for the desktop backend to support all the major desktop operating systems, such as Windows, Linux, and Mac OS X.

For more details, check out the official LWJGL website http://www.lwjgl.org/.

Android

Google frequently releases and updates its official Android SDK. This represents the foundation for LibGDX to support Android in the form of a backend.

There is an API guide available, which explains everything the Android SDK has to offer to Android developers. You can find this at http://developer.android.com/guide/components/index.html.

WebGL

The WebGL support is one of the latest additions to the LibGDX framework. This backend uses the GWT to translate the Java code into JavaScript and SoundManager2 (SM2) among others in order to add a combined support for HTML5, WebGL, and audio playback. Note that this backend requires a WebGL-capable web browser to run the application.

The following are some useful links that will help you get a detailed description:

- Check out the official website of GWT at https://developers.google.com/web-toolkit/
• Check out the official website of SM2 at http://www.schillmania.com/projects/soundmanager2/
• Check out the official website of WebGL at http://www.khronos.org/webgl/
• There is also a list of unresolved issues at https://github.com/libgdx/libgdx/blob/master/backends/gdx-backends-gwt/issues.txt that you might want to check out

**RoboVM (iOS backend)**

The goal of the RoboVM open source project is to bring Java and other JVM languages to iOS. RoboVM's ahead-of-time compiler translates the Java bytecode into a native ARM or x86 machine code that runs directly on the target CPU without being interpreted. The runtime is based on Android's runtime classes and includes a Java to Objective-C bridge, which makes it easy to use the native Cocoa Touch APIs from Java.

You can find more about RoboVM at http://www.robovm.com, and you can check the currently working code at https://github.com/robovm/robovm.

![RoboVM](image)

**LibGDX core modules**

LibGDX provides six core modules that allow you to access various parts of the system your application will run on. What makes these modules so great for you as a developer is that they provide you with a single **Application Programming Interface (API)** to achieve the same effect on more than just one platform. This is extremely powerful because you can now focus on your own application and do not have to bother with the specialties that each platform inevitably brings, including the nasty little bugs that might require tricky workarounds. This is all going to be transparently handled in a straightforward API, which is categorized into logic modules and is globally available anywhere in your code as every module is accessible as a static field in the Gdx class.

LibGDX allows you to create multiple code paths for per-platform decisions. For example, you can increase the complexity of a desktop game as desktops have a lot more computing power than mobile devices.
The application module
The application module can be accessed through Gdx.app. It gives you access to the logging facility, a method to shutdown gracefully, persist data, query the Android API version, query the platform type, and query the memory usage.

Logging
LibGDX employs its own logging facility. You can choose a log level to filter what should be printed to the platform's console. The default log level is LOG_INFO. You can use a settings file and/or change the log level dynamically at runtime using the following code:

```java
Gdx.app.setLogLevel(Application.LOG_DEBUG);
```

The available log levels are as follows:
- LOG_NONE: This prints no logs and the logging is completely disabled
- LOG_ERROR: This prints error logs only
- LOG_INFO: This prints error and info logs
- LOG_DEBUG: This prints error, info, and debug logs

To write an info, debug, or error log to the console, use the following listings:

```java
Gdx.app.log("MyDemoTag", "This is an info log.");
Gdx.app.debug("MyDemoTag", "This is a debug log.");
Gdx.app.error("MyDemoTag", "This is an error log.");
```

Shutting down gracefully
You can tell LibGDX to shut down the running application. The framework will then stop the execution in the correct order as soon as possible and completely deallocate any memory that is still in use, freeing both Java and the native heap. Use the following listing to initiate a graceful shutdown of your application:

```java
Gdx.app.exit();
```

You should always do a graceful shutdown when you want to terminate your application. Otherwise, you will risk creating memory leaks, which is a really bad thing. On mobile devices, memory leaks will probably have the biggest negative impact due to their limited resources. Note that in an Android device, it will call the `pause()` and `dispose()` functions sometime later and won't immediately finish the application.
Persisting data

If you want your data to persist after exit, you should use the `Preferences` class. It is merely a dictionary or a hash map data type that stores multiple key-value pairs in a file. LibGDX will create a new preferences file on the fly if it does not exist. You can have several preference files using unique names in order to split up data into categories. To get access to a preference file, you need to request a `Preferences` instance by its filename as follows:

```java
Preferences prefs = Gdx.app.getPreferences("settingsprefs");
```

To write a (new) value, you have to choose a key under which the value should be stored. If this key already exists in a preferences file, it will be overwritten. Do not forget to call `flush()` afterwards, as shown in the following code, to persist the data, or else all the changes will be lost:

```java
prefs.putInteger("sound_volume", 100); // volume @ 100%
prefs.flush();
```

Persisting data needs a lot more time than just modifying values in memory (without flushing). Therefore, it is always better to modify as many values as possible before a final `flush()` method is executed.

To read back a certain value from the preferences file, you need to know the corresponding key. If this key does not exist, it will be set to the default value. You can optionally pass your own default value as the second argument (for example, in the following listing, 50 is the default sound volume):

```java
int soundVolume = prefs.getInteger("sound_volume", 50);
```

Querying the Android API level

On Android, you can query the Android API level that allows you to handle things differently for certain versions of the Android OS. Use the following listing to find out the version:

```java
Gdx.app.getVersion();
```

On platforms other than Android, the version returned is always 0.
Cross-platform Development – Build Once, Deploy Anywhere

Querying the platform type
You might want to write a platform-specific code where it is necessary to know the current platform type. The following example shows how it can be done:

```java
switch (Gdx.app.getType()) {
    case Desktop:
        // Code for Desktop application
        break;
    case Android:
        // Code for Android application
        break;
    case WebGL:
        // Code for WebGL application
        break;
    case iOS:
        // Code for IOS application
        break;
    default:
        // Unhandled (new?) platform application
        break;
}
```

Querying the memory usage
You can query the system to find out its current memory footprint of your application. This might help you find excessive memory allocations that could lead to application crashes. The following functions return the amount of memory (in bytes) that is in use by the corresponding heap:

```java
long memUsageJavaHeap = Gdx.app.getJavaHeap();
long memUsageNativeHeap = Gdx.app.getNativeHeap();
```

Multithreading
When our game is created, LibGDX creates a separate thread called the **Main loop thread** and OpenGL context is attached to it. The entire event processing or rendering happens within this thread and not in the UI thread. Hence to pass data to the rendering thread from another thread, we use `Application.postRunnable()`. This will run the code in the `Runnable` function in the rendering thread in the next frame, as shown in the following code:

```java
Gdx.app.postRunnable(new Runnable() {
    @Override
    public void run() {
```
The graphics module
The graphics module can be accessed either through `Gdx.getGraphics()` or using the shortcut variable `Gdx.graphics`.

Querying delta time
Query LibGDX for the time span between the current and the last frame in seconds by calling `Gdx.graphics.getDeltaTime()`.

Querying display size
Query the device's display size returned in pixels by calling `Gdx.graphics.getWidth()` and `Gdx.graphics.getHeight()`.

Querying the frames per second (FPS) counter
Query a built-in frame counter provided by LibGDX to find the average number of frames per second by calling `Gdx.graphics.getFramesPerSecond()`.

The audio module
The audio module can be accessed either through `Gdx.getAudio()` or using the shortcut variable `Gdx.audio`.

Sound playback
To load sounds for playback, call `Gdx.audio.newSound()`.

The supported file formats are WAV, MP3, and OGG. However, for the iOS version, OGG is not supported. There is an upper limit of 1 MB for the decoded audio data. Consider the sounds to be short effects such as bullets or explosions so that the size limitation is not really an issue.
Music streaming
To stream music for playback, call `Gdx.audio.newMusic()`. The supported file formats are WAV, MP3, and OGG. However, the iOS version currently supports the WAV and MP3 formats only.

The input module
The input module can be accessed either through `Gdx.getInput()` or using the shortcut variable `Gdx.input`.

In order to receive and handle the input properly, you should always implement the `InputProcessor` interface and set it as the global handler for the input in LibGDX by calling `Gdx.input.setInputProcessor()`.

Reading the keyboard/touch/mouse input
Query the system for the last x or y coordinate in the screen coordinates, where the screen origin is at the top-left corner by calling either `Gdx.input.getX()` or `Gdx.input.getY()`. The different conditions are as follows:

- To find out whether the screen is touched either by a finger or by mouse, call `Gdx.input.isTouched()`.
- To find out whether the mouse button is pressed, call `Gdx.input.isButtonPressed()`.
- To find out whether the keyboard key is pressed, call `Gdx.input.isKeyPressed()`.

Reading the accelerometer
Query the accelerometer for its value on the x axis by calling `Gdx.input.getAccelerometerX()`. Replace X in the method's name with Y or Z to query the other two axes. Be aware that there will be no accelerometer present on a desktop, so LibGDX always returns 0.

Starting and canceling vibrator
On Android, you can let the device vibrate by calling `Gdx.input.vibrate()`. A running vibration can be canceled by calling `Gdx.input.cancelVibrate()`.
Catching Android's soft keys
You might want to catch Android's soft keys to add an extra handling code for them. If you want to catch the back button, call `Gdx.input.setCatchBackKey(true)`, and if you want to catch the menu button, call `Gdx.input.setCatchMenuKey(true)`.

On a desktop where you have a mouse pointer, you can tell LibGDX to catch it so that you get a permanent mouse input without having the mouse ever leave the application window. To catch the mouse cursor, call `Gdx.input.setCursorCatched(true)`.

The files module
The files module can be accessed either through `Gdx.getFiles()` or using the shortcut variable: `Gdx.files`.

Getting an internal file handle
You can get a file handle for an internal file by calling `Gdx.files.internal()`. An internal file is relative to the `assets` folder on the Android and WebGL platforms. On a desktop, it is relative to the root folder of the application.

Getting an external file handle
You can get a file handle for an external file by calling `Gdx.files.external()`. An external file is relative to the SD card on the Android platform. On a desktop, it is relative to the user's home folder. Note that this is not available for WebGL applications.

The network module
The network module can be accessed either through `Gdx.getNet()` or using the shortcut variable: `Gdx.net`.

HTTP requests
You can make HTTP requests by calling `Gdx.net.sendHttpRequest()` or cancel them by calling `Gdx.net.cancelHttpRequest()`.
Client/server sockets
You can create client/server sockets by calling either `Gdx.net.newClientSocket()` or `Gdx.net.newServerSocket()`.

Opening a URI in a web browser
To open a Uniform Resource Identifier (URI) in the default web browser, call `Gdx.net.openURI(URI)`.

LibGDX’s application life cycle and interface
The application life cycle in LibGDX is a well-defined set of distinct system states. The list of these states is pretty short: create, resize, render, pause, resume, and dispose.

LibGDX defines an `ApplicationListener` interface that contains six methods, one for each system state. The following code listing is a copy that is directly taken from LibGDX’s sources. For the sake of readability, all comments have been stripped:

```java
public interface ApplicationListener {
    public void create ();
    public void resize (int width, int height);
    public void render ();
    public void pause ();
    public void resume ();
    public void dispose ();
}
```

All you need to do is implement these methods in your main class of the shared game code project. LibGDX will then call each of these methods at the right time.

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.
The following diagram visualizes the LibGDX’s application life cycle:

Note that a full and dotted line basically has the same meaning in the preceding diagram. They both connect two consecutive states and have a direction of flow indicated by a little arrowhead on one end of the line. A dotted line additionally denotes a system event.

When an application starts, it will always begin with `create()`. This is where the initialization of the application should happen, such as loading assets into memory and creating an initial state of the game world. Subsequently, the next state that follows is `resize()`. This is the first opportunity for an application to adjust itself to the available display size (width and height) given in pixels.

Next, LibGDX will handle system events. If no event has occurred in the meanwhile, it is assumed that the application is (still) running. The next state would be `render()`. This is where a game application will mainly do the following two things:

- Update the game world model
- Draw the scene on the screen using the updated game world model

Afterwards, a decision is made on which the platform type is detected by LibGDX. On a desktop or in a web browser, the displaying application window can be virtually resized at any time. LibGDX compares the last and current sizes on every cycle so that `resize()` is only called if the display size is changed. This makes sure that the running application is able to accommodate a changed display size.
Now, the cycle starts over by handling (new) system events once again. Another system event that can occur during runtime is the exit event. When it occurs, LibGDX will first change to the `pause()` state, which is a very good place to save any data that would be lost otherwise, after the application is terminated. Subsequently, LibGDX changes to the `dispose()` state where an application should do its final cleanup to free all the resources that it is still using.

This is also almost true for Android, except that `pause()` is an intermediate state that is not directly followed by the `dispose()` state at first. Be aware that this event might occur anytime during an application runtime when the user has pressed the Home button or if there is an incoming phone call in the meanwhile. In fact, as long as the Android operating system does not need the occupied memory of the paused application, its state will not be changed to `dispose()`. Moreover, it is possible that a paused application might receive a `resume` system event, which in this case would change its state to `resume()`, and it would eventually arrive at the system event handler again.

### Starter classes

A starter class defines the entry point (starting point) of a LibGDX application. It is specifically written for a certain platform. Usually, these kinds of classes are very simple and mostly consist of not more than a few lines of code to set certain parameters that apply to the corresponding platform. Think of them as a kind of bootup sequence for each platform. Once booting is finished, the LibGDX framework hands over control from the starter class (for example, the `demo-desktop project`) to your shared application code (for example, the `demo/demo-core project`) by calling different methods from the `ApplicationListener` interface that the `MyDemo` class implements. Remember that the `MyDemo` class is where the shared application code begins.

We will now take a look at each of the starter classes that were generated during the project setup.

### Running the demo application on a desktop

The starter class for the desktop application is called `Main.java`. The following listing is `Main.java` from the `demo-desktop project`:

```java
package com.packtpub.libgdx.demo;
import com.badlogic.gdx.backends.lwjgl.LwjglApplication;
import com.badlogic.gdx.backends.lwjgl.LwjglApplicationConfiguration;
public class Main {
```
public static void main(String[] args) {

    LwjglApplicationConfiguration cfg = new LwjglApplicationConfiguration();
    cfg.title = "demo";
    cfg.width = 480;
    cfg.height = 320;
    new LwjglApplication(new MyDemo(), cfg);
}

In the preceding code listing, you can see the Main class: a plain Java class without
the need to implement an interface or inherit from another class. Instead, a new
instance of the LwjglApplication class is created. This class provides a couple
of overloaded constructors to choose from. Here, we pass a new instance of the
MyDemo class as the first argument to the constructor. Optionally, an instance of
the LwjglApplicationConfiguration class can be passed as the second argument.
The configuration class allows you to set every parameter that is configurable for
a LibGDX desktop application. In this case, the window title is set to demo, and the
window's width and height is set to 480 by 320 pixels.

This is all you need to write and configure a starter class for a desktop project.
Let's try to run the application now. To do this, right-click on the demo-desktop
project in Project Explorer in Eclipse and then select the Java Application option
from the Run As menu. Eclipse might ask you to select the Main class when you
do this for the first time. Simply select the Main class, and also check whether the
correct package name (com.packtpub.libgdx.demo) is displayed next to it, as
shown in the following screenshot:
Cross-platform Development – Build Once, Deploy Anywhere

For those who use Gradle to set up the project, remember that the starter class of the desktop project will be DesktopLauncher.java and the correct package name will be com.packtpub.libgdx.demo.desktop.

The desktop application should now be up and running on your computer. If you are working on Windows, you should see the following window:

For Gradle users, this image will be displayed:
Running the demo application on Android

The starter class for the Android application is called MainActivity.java. For a Gradle-based project, the starter class will be AndroidLauncher.java.

The following listing is MainActivity.java from demo-android:

```java
package com.packtpub.libgdx.demo;
import android.os.Bundle;
import com.badlogic.gdx.backends.android.AndroidApplication;
import com.badlogic.gdx.backends.android.AndroidApplicationConfiguration;
public class MainActivity extends AndroidApplication {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        AndroidApplicationConfiguration cfg = new AndroidApplicationConfiguration();
        initialize(new MyDemo(), cfg);
    }
}
```

In the preceding code listing, you can see the MainActivity class that is inherited from the AndroidApplication class. This is how LibGDX encapsulates tasks such as creating a so-called activity that registers handlers to process touch input, read sensor data, and much more. What is left to do for you is to create an instance of a class that implements the ApplicationListener interface. In this case, it is an instance of the MyDemo class. The instances of MyDemo and AndroidApplicationConfiguration are passed as arguments to the initialize() method. If you are interested in the latest development of Android hardware statistics, be sure to check out the Dashboards section on the official Android developer website (http://developer.android.com/about/dashboards/index.html#OpenGL).
The following screenshot of the OpenGL statistics was taken in May 2014 from the preceding mentioned website:

![OpenGL Statistics Screenshot](image)

Note that GLES 1.1 is nearly zero. So what's the big deal about GLES 2.0? A better question to ask would be whether you plan to use shaders in your application. If this is the case, opt for GLES 2.0.

LibGDX has now removed the support for GLES 1.0, so the default OpenGL version is 2.0.

In any other case, there will be no real benefit except being able to use Non-Power-Of-Two (NPOT) textures—arbitrarily sized textures that do not equal to widths or heights representable by the formula $2^n$, such as 32 x 32, 512 x 512, and 128 x 1024.

NPOT textures are not guaranteed to work on all devices. For example, Nexus One ignores NPOT textures. Also, it might cause performance penalties on some hardware, so it is best to avoid using this feature at all. In Chapter 4, Gathering Resources, you will learn about a concept called Texture Atlas. This will allow you to use arbitrarily sized textures even when you are not using GLES2.
Additionally, on Android, you will have to take care of a manifest file that defines a huge list of parameters to configure the application. If you are not yet familiar with Android's manifest file, read the official documentation at http://developer.android.com/guide/topics/manifest/manifest-intro.html.

The following listing is `AndroidManifest.xml` from demo-android:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.packtpub.libgdx.demo"
  android:versionCode="1"
  android:versionName="1.0">
  <uses-sdk android:minSdkVersion="8"
    android:targetSdkVersion="19"/>
  <uses-feature android:glEsVersion="0x00020000"
    android:required="true"/>
  <application
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name">
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name"
      android:screenOrientation="landscape"
      android:configChanges="keyboard|keyboardHidden|orientation">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

There will be an error displayed in `android:configChanges` after changing to Android API level 8, as shown in the following screenshot:
Cross-platform Development – Build Once, Deploy Anywhere

This is because the value `screenSize` is not supported by the API Level 8. Just remove this value for our project. To know more about this `android:configChanges` element and other elements, visit http://developer.android.com/guide/topics/manifest/activity-element.html.

The following short (and incomplete) list is meant to give you a quick idea of what could be defined in the manifest file:

- `minSdkVersion`: This is the minimum API Level required for the application to run. Devices running with lower API Levels will not be able to run this application; if left undeclared, an API Level of 1 is assumed, which might cause your app to crash at runtime when trying to access unavailable APIs.
- `targetSdkVersion`: This is the API Level the application targets. This is used for forward compatibility, where later API Levels can change the behavior of the API that might break old applications. This specification does not prevent the application from running on devices with lower API Levels down to `minSdkVersion`. If left undeclared, its value is set equal to `minSdkVersion`.
- `icon`: This is the application's icon.
- `name`: This is the main class of the application (or the main activity). Note that in terms of LibGDX, this will be the starter class for Android.
- `label`: This is the application's name shown next to the application icon and in the title bar.
- `screenOrientation`: This defines the display orientation of the application. The usual values are `portrait` (tall) and `landscape` (wide). See the documentation for more details.

Another crucial part of the manifest file is the correct definition of the permissions that the application should request when a user wants to install it on a device.

> Make sure that you will never request unnecessary permissions and put as much information as required into the description text of your application. Users are extremely suspicious and justifiably so when it comes to the list of requested permissions. It is not 100 percent clear for which reason an application needs a certain permission.

For an introduction and much more detail on the topic of permissions on Android, refer to the official documentation at http://developer.android.com/guide/topics/security/permissions.html.
Now, let's try to run the application on a real, physical device. First, make sure that your Android device is connected via a USB cable and is set up for development. To set up your Android device, follow the instructions at http://developer.android.com/tools/device.html.

Now, right-click on the demo-android project in Project Explorer in Eclipse and select the Android Application option from the Run As menu.

The Android application should now be installed and be happily running on your Android device. The following image is of the application running on an HTC Desire HD:

![Image of the application running on an HTC Desire HD]

With regards to the Android emulator that comes with the Android SDK, just a few final words, do not use it!

Emulators cannot accurately reflect how a device responds, so instead of using an emulator, it is highly recommended to try and test your applications on as many real devices as you can get your hands on.

## Running the demo application in a WebGL-capable web browser

The starter class for the WebGL application is called GwtLauncher.java. The following listing is GwtLauncher.java from demo-html:

```java
package com.packtpub.libgdx.demo.client;
import com.packtpub.libgdx.demo.MyDemo;
```
import com.badlogic.gdx.ApplicationListener;
import com.badlogic.gdx.backends.gwt.GwtApplication;
import com.badlogic.gdx.backends.gwt.GwtApplicationConfiguration;

public class GwtLauncher extends GwtApplication {
    @Override
    public GwtApplicationConfiguration getConfig () {
        GwtApplicationConfiguration cfg = new
            GwtApplicationConfiguration(800, 480);
        return cfg;
    }
    @Override
    public ApplicationListener getApplicationListener () {
        return new MyDemo();
    }
}

In the preceding code listing, you can see the GwtLauncher class that is inherited from the GwtApplication class. LibGDX encapsulates GWT and only requires you to implement the two abstract methods, getConfig() and getApplicationListener(). The getConfig() method returns an instance of the GwtApplicationConfiguration class. In this case, the window's width and height are directly passed on to its constructor. The getApplicationListener() method returns an instance of a class that implements the ApplicationListener interface, which is the MyDemo class in the preceding code.

Additionally, GWT is organized in so-called modules that bundle together all the configuration settings. In this case, we only have one module called MyDemo.gwt.xml. It defines the source path where GWT should look for Java source files, in this case, com/packtpub/libgdx/demo. These source files will then be cross-compiled by GWT to optimize the JavaScript code that is runnable on all major web browsers.

The following listing is MyDemo.gwt.xml from demo project:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<module>
    <source path="com/packtpub/libgdx/demo" />
</module>
```
Let's try to run the application now. To do this, right-click on the demo-html project in Project Explorer in Eclipse and then select the Web Application option in the Run As menu. A new tab titled Development Mode will open at the bottom. Simply do what the description preceding the URL says and double-click on it. Your default browser should launch shortly after this. Then try to load the website that is hosted on your local machine right now. Hence, the URL points to 127.0.0.1, the infamous IPv4 loopback address that is just another fancy name for this device or computer:

Keep in mind that using the URL suggested by Eclipse will run your WebGL application in debug mode, which is excruciatingly slow for most games. Just remove everything after the question mark in the URL to run your application in normal mode. The resulting URL should look like http://127.0.0.1:8888/index.html.

You might be asked to install the Google Web Toolkit Developer Plugin for your web browser to use the Development Mode, as shown in the following screenshot. You need to install it to develop your local machine.
After the plugin has been successfully installed, you will see the following window:

If you want to run this application on a real web server and share it with other users on the Internet, you will have to cross-compile the project first. This is a pretty straightforward process. Simply right-click on the demo-html project in Project Explorer in Eclipse and then select the GWT Compile option from the Google menu, as shown here:

A window with the title GWT Compile will open. Here, you can choose a log level to narrow down on certain messages such as errors only. Keep the default settings for now and click on Compile to begin the cross-compile process, as shown here:
The compilation process is quite lengthy compared to all the other ones shown in this book. It took over two minutes to finish on an Intel Core i7 (3.4GHz) processor. A good moment to exercise your patience!
Once the compilation is finished, go to the war subfolder in the demo-html project, as shown in the following screenshot:

You can now upload everything to your web server that is contained in this folder except the WEB-INF folder, which is not needed. Now, you or anyone else can open the URL to your web server and enjoy your LibGDX cross-platform application in a WebGL-capable web browser without having to install any plugin for it to work.

### Running the demo application on an iOS device

The starter class for iOS application is RobovmLauncher.java. For Gradle, it is IOSLauncher.java. The following listing is from RobovmLauncher.java in demo-robovm:

```java
package com.packtpub.libgdx.demo;

import org.robovm.apple.foundation.NSAutoreleasePool;
import org.robovm.apple.uikit.UIApplication;
import com.badlogic.gdx.backends.iosrobovm.IOSApplication;
import com.badlogic.gdx.backends.iosrobovm.IOSApplicationConfiguration;

public class RobovmLauncher extends IOSApplication.Delegate {
    @Override
    protected IOSApplication createApplication() {
        IOSApplicationConfiguration config = new IOSApplicationConfiguration();
        config.orientationLandscape = true;
        config.orientationPortrait = false;
        return new IOSApplication(config);
    }
}
```
return new IOSApplication(new MyDemo(), config);
}

public static void main(String[] argv) {
    NSAutoreleasePool pool = new NSAutoreleasePool();
    UIApplication.main(argv, null, RobovmLauncher.class);
    pool.close();
}
}

In the preceding code, you can see the RobovmLauncher class that is inherited from the IOSApplication.Delegate class. Here is where LibGDX encapsulates tasks and registers handlers to process touch input, and other sensor data, and much more. The instances of MyDemo and IOSApplicationConfiguration are passed as arguments to the IOSApplication function.

In Android, we saw the AndroidManifest.xml file that specifies the characteristics, permissions, and other features of our Demo-Android app. Similarly, our iOS app has Info.plist.xml to hold such details. Before explaining Info.plist.xml, let's see the robovm.properties and robovm.xml file.

The following code is taken from robovm.properties file in our demo-robovm project:

```properties
app.version=1.0
app.id=com.packtpub.libgdx.demo
app.mainclass=com.packtpub.libgdx.demo.RobovmLauncher
app.executable=MyDemo
app.build=1
app.name=MyDemo
```

This brief file contains, as the statements indicate, the app version, app ID, main class, executable, build number, and name of app. These values will be used in the Info.plist.xml file:

The following listing is taken from robovm.xml in our demo-robovm project:

```xml
<config>
    <executableName>${app.executable}</executableName>
    <mainClass>${app.mainclass}</mainClass>
    <os>ios</os>
    <arch/thumbv7</arch>
    <target>ios</target>
    <iosInfoPList>Info.plist.xml</iosInfoPList>
    <resources>
        <resource>
            <directory>../android/assets</directory>
```

```xml```
Cross-platform Development – Build Once, Deploy Anywhere

This file holds the important link: the path to the assets folder in the demo-android project. Under the <resource> key, the path to the Android assets folder is set. However, we need iOS-specific icons and splash images, and we don’t need to put this in the Android assets folder (believe me, you don’t want to increase your Android APK size with unwanted data). Instead, we put it in the folder data inside the demo-robovm project and include the path under the <resources> key in robovm.xml.

Now comes the crucial part: the Info.plist file. Every iOS app contains the Info.plist file and it holds crucial information about the characteristics, permissions, and other features about the app. In our RoboVM version, it is named Info.plist.xml. The following code is taken from the Info.plist.xml file in our demo-robovm project:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
```
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>en</string>
    <key>CFBundleDisplayName</key>
    <string>${app.name}</string>
    <key>CFBundleExecutable</key>
    <string>${app.executable}</string>
    <key>CFBundleIdentifier</key>
    <string>${app.id}</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>${app.name}</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleShortVersionString</key>
    <string>${app.version}</string>
    <key>CFBundleSignature</key>
    <string>????</string>
    <key>CFBundleVersion</key>
    <string>${app.build}</string>
    <key>LSRequiresIPhoneOS</key>
    <true/>
    <key>UIStatusBarHidden</key>
    <true/>
    <key>UIViewControllerBasedStatusBarAppearance</key>
    <false/>
    <key>UIDeviceFamily</key>
    <array>
        <integer>1</integer>
        <integer>2</integer>
    </array>
    <key>UIRequiredDeviceCapabilities</key>
    <array>
        <string>armv7</string>
    </array>
    <key>UISupportedInterfaceOrientations</key>
    <array>
        <string>UIInterfaceOrientationLandscapeLeft</string>
        <string>UIInterfaceOrientationLandscapeRight</string>
    </array>
    <key>UISupportedInterfaceOrientations-ipad</key>
    <array>
        <string>UIInterfaceOrientationLandscapeLeft</string>
    </array>
</dict>
The following short (and incomplete) list will give you a quick idea of what the Info.plist keys means:

- **UISupportedInterfaceOrientations**: This key is used to set the allowed device orientations:
  - For iPads, it is `UISupportedInterfaceOrientations~ipad`
  - For iPhones and iPads, the values for these keys are:
    - `UIInterfaceOrientationPortrait`
    - `UIInterfaceOrientationPortraitUpsideDown`
    - `UIInterfaceOrientationLandscapeRight`
    - `UIInterfaceOrientationLandscapeLeft`

- **UIRequiredDeviceCapabilities**: This key lets you declare the hardware or specific capabilities that your app needs in order to run. For example, Wi-Fi, Bluetooth, accelerometer, open GLES 2.0, and so on.

- **CFBundleName**: This is the name of the application as specified in the `robovm.properties` file.

- **CFBundleIdentifier**: This is the unique identifier of the application as specified in the `robovm.properties` file. For our demo-robovm project, it is `CFBundleIdentifier com.packtbub.libgdx.demo`.

- **CFBundleIconFiles**: These are the application icons.
For more information on the topic of the `Info.plist` keys, check out the official documentation at https://developer.apple.com/library/mac/documentation/general/Reference/InfoPlistKeyReference/Articles/iPhoneOSKeys.html#

For more about the device capabilities, check out the official documentation at https://developer.apple.com/library/mac/documentation/general/Reference/InfoPlistKeyReference/Articles/iPhoneOSKeys.html#

Now, right-click on the `demo-robovm` project in **Project Explorer** in Eclipse and select the **iOS Device App** option in the **Run As** menu.

Remember that to execute an iOS application, you need a Mac machine.

The iOS application should now be installed as an application icon and should be happily running on your iOS device. The following image is of the application running on an IPad 3:
The demo application – time for code
In this section, we will take a closer look at the actual code of the demo project. Thereafter, we will do some simple modifications to the code and also use the debugger.

Inspecting an example code of the demo application
Let's take a first look at the generated code of MyDemo.java from the demo project.

The following code snippet shows the class definition:

```java
public class MyDemo implements ApplicationListener {
    // ...
}
```

As you can see, the MyDemo class implements the ApplicationListener interface. The MyDemo class from Gradle project (demo-core) produces a quite different code as follows:

```java
public class MyDemo extends ApplicationAdapter {
    //...
}
```

Here, the ApplicationAdapter is an abstract class that implements the ApplicationListener interface. Before we move on to the implementation details of the interface, we will spend some time on the remaining part of this class.

You will find a definition of the four member variables, each with a class provided by LibGDX:

```java
Private OrthographicCamera camera;
Private SpriteBatch batch;
private Texture texture;
private Sprite sprite;
```

Here is a brief explanation of the classes from the preceding code listing to give you the basic background knowledge for the code inspection that will follow shortly.

The camera variable is of the class type OrthographicCamera. We will use the orthographic camera to display our 2D scenes. The camera is the player's view of the actual scene in the game, which is defined by a certain width and height (also called viewport).

The batch variable is of the class type SpriteBatch. This is where you send all your drawing commands to LibGDX. Beyond the ability of this class to draw images, it is also capable of optimizing the drawing performance under certain circumstances.

The texture variable is of the class type Texture. It holds a reference to the actual image; the texture data that is stored in memory at runtime.

The sprite variable is of the class type Sprite. It is a complex data type that contains lots of attributes to represent a graphical object that has a position in 2D space, width, and height. It can also be rotated and scaled. Internally, it holds a reference to a TextureRegion class that in turn is a means to cut out a certain portion of a texture.

Now that we have a basic knowledge of the involved data types, we can advance to the implementation details of the ApplicationListener interface.

In the MyDemo class, the only methods containing code are create(), render(), and dispose(). The remaining three methods are left empty, which is just fine.

**The create() method**

The create() method contains the initialization code to prepare the application on startup, as shown in the following code snippet:

```java
@Override
public void create() {

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

    camera = new OrthographicCamera(1, h/w);
    batch = new SpriteBatch();

    texture = new Texture(Gdx.files.internal("data/libgdx.png"));
    texture.setFilter(TextureFilter.Linear, TextureFilter.Linear);

    TextureRegion region =
        newTextureRegion(texture, 0, 0, 512, 275);

    sprite = new Sprite(region);
}
```
sprite.setSize(0.9f, 0.9f * sprite.getHeight() / sprite.getWidth());
sprite.setOrigin(sprite.getWidth()/2, sprite.getHeight()/2);
sprite.setPosition(-sprite.getWidth()/2, -sprite.getHeight()/2);
}

At first, the graphics module is queried to return the width and height of the display (for example, a desktop window or the screen of an Android device) and calculate an appropriate width and height for the field of view of the camera. Then, a new instance of SpriteBatch is created so that images can be drawn and made visible with the camera. The next step is to load a texture using the files module to get a file handle to data/libgdx.png.

Gradle users will find only two lines of code in the create() method and only four lines of code in the render() method. The assets folder in the demo-android project will contain only an image labeled badlogic.jpg. Read and understand this code; the complete code is given at the end of this section.

The loaded texture looks like the following screenshot:
As you can see, there is a lot of empty space in this screenshot. In order to be able to use the filled part of this screenshot only, a new instance of `TextureRegion` is created. It references the previously loaded texture that contains the full image and has the additional information to cut all the pixels starting from \((0, 0)\) to \((512, 275)\). These two points describe a rectangle starting at the top-left corner of the image with a width and height of 512 by 275 pixels. Finally, a sprite is created using the information of the previously created texture region. The sprite's size is set to 90 percent of its original size. The sprite's origin is set to half of its width and height to move the origin to its center. Eventually, the position is set to the negative half of the sprite's width and height so that the sprite moves to the center of the scene.

LibGDX uses a coordinate system that has its origin \((0, 0)\) at the bottom-left corner. This means that the positive \(x\) axis points to the right-hand side, while the positive \(y\) axis points upwards.

### The `render()` method

The `render()` method contains the commands to render a scene on screen, as shown here:

```java
@Override
public void render() {
    Gdx.gl.glClearColor(1, 1, 1, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    batch.setProjectionMatrix(camera.combined);
    batch.begin();
    sprite.draw(batch);
    batch.end();
}
```

The first two lines call the low-level OpenGL methods to set the clear color to a solid white, and then execute the clear screen command.

Next, the projection matrix of the sprite batch is set to the camera's combined projection and view matrix. You do not have to understand what this means in detail at the moment. It basically just means that every following drawing command will behave according to the rules of an orthographic projection, or simply put, drawing will be done in 2D space using the position and bounds of the given camera.

The `begin()` and `end()` methods will always have to appear in pairs and should not be nested or there will be errors. The actual drawing of the sprite is accomplished by calling the `draw()` method of the sprite to draw and pass the instance of the sprite batch.
The dispose() method

The dispose() method is the place where you clean up and free all resources that are still in use by an application, as shown here:

```java
@Override
public void dispose() {
    batch.dispose();
    texture.dispose();
}
```

There is an interface called Disposable that is implemented by every LibGDX class that allocates resources (that is, memory) and can be easily deallocated by calling the corresponding dispose method. In the preceding code, this is done for the sprite batch and the loaded texture.

The following is a complete listing of the MyDemo.java source file from the demo project:

```java
package com.packtpub.libgdx.demo;

import com.badlogic.gdx.ApplicationListener;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.Net.HttpRequest;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.Texture.TextureFilter;
import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.graphics.g2d.TextureRegion;

public class MyDemo implements ApplicationListener {
    private OrthographicCamera camera;
    private SpriteBatch batch;
    private Texture texture;
    private Sprite sprite;

    @Override
    public void create() {
        float w = Gdx.graphics.getWidth();
```
float h = Gdx.graphics.getHeight();
camera = new OrthographicCamera(1, h/w);
batch = new SpriteBatch();
texture = new Texture(Gdx.files.internal("data/libgdx.png"));
texture.setFilter(TextureFilter.Linear,
    TextureFilter.Linear);
TextureRegion region = new TextureRegion(texture, 0, 0,
    512, 275);
sprite = new Sprite(region);
sprite.setSize(0.9f, 0.9f * sprite.getHeight() /
    sprite.getWidth());
sprite.setOrigin(sprite.getWidth()/2,
    sprite.getHeight()/2);
sprite.setPosition(-sprite.getWidth()/2, -
    sprite.getHeight()/2);
}

@Override
public void dispose() {
batch.dispose();
texture.dispose();
}

@Override
public void render() {
    Gdx.gl.glClearColor(1, 1, 1, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    batch.setProjectionMatrix(camera.combined);
    batch.begin();
sprite.draw(batch);
    batch.end();
}

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

@Override
Cross-platform Development – Build Once, Deploy Anywhere

```java
public void pause() {
}

@Override
public void resume() {
}
```

Gradle users should copy this code to the `MyDemo` class. We need this code to do a simple experimentation with code hot swapping, which is coming in the next section. Although you don't have the `libgdx.png` file, you can use any standard paint tools to create a placeholder image of size 512 x 512, or download the `libgdx.png` file from the example project in the provided code bundle. Then, go to the `assets` directory in the `demo-android` project, create a `data` folder, and save the `libgdx.png` file in it.

Having fun with the debugger and Code Hot Swapping

In this section, we are going to use the debugger to take a look inside the `demo` project at runtime. To do this, we first set a breakpoint where the execution of the application should be halted so that we can easily inspect the current state. Open the `MyDemo.java` source file in Eclipse and set a breakpoint at the line where a new instance of `SpriteBatch` is created, as shown here:

```java
camera = new OrthographicCamera(1, h/w);
batch = new SpriteBatch();
texture = new Texture(Gdx.files.internal("data/libgdx.png"));
texture.setFilter(TextureFilter.Linear, TextureFilter.Linear);
```

Double-click on the shaded, empty space at the very left-hand side of the editor window in Eclipse to set or remove already existing breakpoints, which will insert a blue dot to signify the breakpoint, as shown in the preceding screenshot.
Next, right-click on the demo-desktop project in Project Explorer in Eclipse, and then select the Java Application option from the Debug As menu, or press the F11 key on your keyboard. The application should be halted almost directly after the application window becomes visible. Eclipse should have automatically changed to the debugging perspective, which shows lots of extra information about an application running in the debug mode, as shown here:

In the Variables tab, you can now inspect every variable that is within the current scope of execution, for example, the two floating-point variables, w and h, have already been set. You can check this by looking for them in the Variables tab. The correct values of the variables are displayed as 480.0 for w and 320.0 for h. To step through, resume or stop the execution of the application; you can go to the Run menu and choose the appropriate menu item. Choose to resume the application for now.

Let’s try to do code hot swapping now. Make sure that the demo application is currently running and is being executed right now. The following code listing is a modified version of the render() method; the modification is highlighted:

```java
@Override
public void render() {
    Gdx.gl.glClearColor(1, 1, 1, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    batch.setProjectionMatrix(camera.combined);
```
Cross-platform Development – Build Once, Deploy Anywhere

```java
batch.begin();
sprite.setRotation(45);
sprite.draw(batch);
batch.end();
}
```

As a result, the following line to your code inside the `MyDemo.java` source file right before `sprite.draw()` is called:

```java
sprite.setRotation(45);
```

It will make the sprite rotate by 45 degrees in a counter-clockwise direction, as shown in the following screenshot. The next step is to save your changes to the source file. What you should see now is that the change you have just made to the code is immediately reflected in the still running application!
For code hot swapping to work, it is necessary that the automatic (re)build feature is enabled. You can quickly check this by going to the Project menu and making sure that the menu item Build Automatically is checked.

You might already sense the possibilities that this great feature enables a developer to do. Just think of a somewhat more complex scene where you are trying to find the best-looking positions for your objects, or you just want to see how it would look with a couple of different settings. It’s a piece of cake with a tool like code hot swapping at your disposal.

Let’s take the preceding example a bit further and make the image rotate continuously.

We will need a variable to store the current rotation value. This value is going to be increased over a period of time. To avoid a possible overflow in rot, we calculate the remainder of the new rotation value divided by 360 degrees. This can be done in an easy way using the modulo operator (\%) to wrap around a certain value.

The rotation is calculated in degrees per second. Afterwards, we set the new rotation value of the sprite and draw it while the rotation value is advanced step by step.

The following listing is the modified code for the rotating image:

```java
private float rot;

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

    batch.setProjectionMatrix(camera.combined);
    batch.begin();
    final float degreesPerSecond = 10.0f;
    rot = (rot + Gdx.graphics.getDeltaTime() *
            degreesPerSecond) % 360;
    sprite.setRotation(rot);
    sprite.draw(batch);
    batch.end();
}
```
Note that some changes cannot be hot swapped into a running application, such as changing method names and introducing new variables in class. In order to reflect these changes, you have to rerun the program. However, in such situations, Eclipse will issue a warning when the code cannot be hot swapped.

Now that we have a changing value for the rotation, let's have some more fun with it and turn the continuous rotation effect into a shake effect.

As the sine (or cosine) function has an oscillating behavior, we can make perfect use of it to make the image shake by a certain amount to the left and right. The amount (amplitude) can be increased and decreased by multiplying it with the answer of the sine function.

The following listing is the modified code for the shaking image:

```java
@Override
public void render() {
    Gdx.gl.glClearColor(1, 1, 1, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    batch.setProjectionMatrix(camera.combined);
    batch.begin();
    float degreesPerSecond = 10.0f;
    rot = (rot + Gdx.graphics.getDeltaTime() * degreesPerSecond) % 360;
    float shakeAmplitudeInDegrees = 5.0f;
    float shake = MathUtils.sin(rot) * shakeAmplitudeInDegrees;
    sprite.setRotation(shake);
    sprite.draw(batch);
    batch.end();
}
```
The following diagram visualizes the effect of both the rotation and the shake:
Summary

In this chapter, you learned a lot about LibGDX and how all the projects of an application work together. We covered LibGDX’s backends, modules, and starter classes. Additionally, we covered what the application life cycle and corresponding interface are and how they are meant to work. The debugger has been used to inspect the demo application at runtime, and furthermore we made use of the JVM Code Hot Swapping feature.

We now know the basics of the LibGDX applications, so we are ready to start developing a real game. We will start at the very beginning of the development cycle step by step. As LibGDX is a framework and not a game engine, we first have to build our own engine. So, we will learn how to create an appropriate program architecture in the next chapter that is suitable to handle our game.
Where to buy this book
You can buy Learning LibGDX Game Development Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.