LibGDX Game Development By Example

LibGDX is a cross-platform game development framework in Java that makes game programming easier and fun to do. It currently supports Windows, Linux, Mac OS X, Android, and HTML5.

This book starts with a simple game through which the game update cycle is explained, including loading textures onto your screen, moving them around, and responding to input. From there, you’ll move on to more advanced concepts such as creating a formal game structure with a menu screen, adding a game screen and loading screen, sprite sheets, and animations. You’ll explore how to introduce a font to optimize text, and with the help of a game that you’ll create, you’ll familiarize yourself with the 2D Tile Map API to create worlds that scroll as the characters move.

Finally, you will discover how to integrate third-party services into games and take a sneak peak at the Social Media API to get a basic understanding of how it fits into the LibGDX ecosystem.

Who this book is written for
This book is intended for those who wish to learn the concepts of game development using LibGDX. An understanding of Java and other programming languages would definitely be helpful, although this is not a must.

What you will learn from this book
- Create and configure a LibGDX project to get started with making games
- Get to grips with a simple game loop that will drive your games
- Manage game assets to reduce code duplication and speed up development
- Pack game assets together into single assets to increase your game's performance
- Display textures on the screen and manipulate them with play input
- Play various types of sounds that a game can generate
- Design and modify a game user interface with LibGDX's built-in tools
- Develop a game that will run across various platforms

Learn how to create your very own game using the LibGDX cross-platform framework

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Let's Get These Snakes Out of This Book!'
- A synopsis of the book’s content
- More information on LibGDX Game Development By Example
James Cook is a Java software developer from London, England. He is currently touring Australia. He has developed software ranging from banking and mobile applications to games. He has worked for large gaming companies, such as Electronic Arts, Playfish, and Plumbee.

During the day, he can be found working on rapid application prototyping for new social casino experiences. However, during the night, he collaborates with his longtime working partner, Dan Smallman, for Super Cookie Games to create games such as Super Bomb Noms, OMG Dancer!, and Betamax—Sherbet Plains.

Super Cookie Games started out as a simple endeavor for James and Dan to learn how to make games and deliver them to Android, iOS, and the Web. Choosing LibGDX as the platform was a key component for the success of this duo, with six different games across three app stores culminating in 50,000 downloads—a testament to LibGDX and its brilliant feature set and ability.
Preface

Video games have been around for over 30 years now and many of us have grown up with some form of experience playing video games. Most of us will want to go on and make our own games. It has never been easier to create your own game and deliver it to the world, whether you want to make a game to share with your friends or want to start a career in the game development industry.

Owing to the ease of development, a whole host of different tools are available to help you create games. Depending on whether you want to develop for desktop, iOS, Android, or HTML5, there are different tools available. However, what if you wanted to develop for all platforms? This is where an awesome framework called LibGDX comes in.

If you are reading this book, you are probably already aware of LibGDX. You may not be proficient in it, and you may not even have used it yet. But that's OK! That's why you are reading this book. We will use LibGDX to create a series of games, each game introducing another set of features of LibGDX. Hopefully, by the end of this book, you will have the foundation to go on and comfortably start making your own games.

What this book covers

Chapter 1, Getting to Know LibGDX, introduces us to LibGDX and helps us to set up our development environment to get ready to create our first ever LibGDX project.

Chapter 2, Let's Get These Snakes Out of This Book!, covers creating our first game using LibGDX—Snake! This also covers the game cycle and how to handle input through LibGDX.
Chapter 3, Making That Snake Slick, covers creating game states, continuing with our Snake game from the previous chapter. This also covers an introduction to the techniques used to help with development and talks about handling different screen sizes and resolutions.

Chapter 4, What the Flap Is the Hype About?, explains how to create our second game, Flappy Bee, our own interpretation of a famous mobile game. Here, we are introduced to handling animations and using LibGDX’s Scene2D to create a GUI menu.

Chapter 5, Making Your Bird More Flightworthy, explores our Flappy Bee game a little further, where we look at handling assets in LibGDX as well as using one of the tools—Heiro—for converting fonts.

Chapter 6, Onto the Next Platform...Game, introduces you to our next game, where we create a simple platform game—Pete the Squirrel! Here, we cover the use of a tile mapping tool—Tiled—and discuss how LibGDX integrates with it.

Chapter 7, Extending the Platform, discusses handling the LibGDX camera to create scrolling levels and introduces playing sounds.

Chapter 8, Why Are All the Birds Angry?, covers our final game in this book. We look at creating our own version of Angry Birds, where we look at how LibGDX and Box2D work together to create an awesome game.

Chapter 9, Even Angrier Birds!, closes our final game by looking at object pooling in LibGDX and showing how it can be used to help with performance and memory management.

Chapter 10, Exporting Our Games to the Platforms, introduces you to how we can use LibGDX to export our awesome games to Android, iOS, and HTML5.

Chapter 11, Third-party Services, covers integrating a platform-specific service into a LibGDX game.
Let's Get These Snakes Out of This Book!

In this chapter, we will start making our first game with the LibGDX framework. We will make a little journey back to one of the very first popular mobile games, Snake. We will start out by looking at LibGDX's update cycle and how textures are handled. Then, we will dive into the world of game making by creating our Snake game!

The following will be covered in this chapter:

- Why Snake?
- The game update cycle
- Introducing the snake
- Making the snake move
- Controlling the snake
- Introducing the collision detection mechanism
- Increasing the length of the snake once the apple is eaten

Why Snake?

Snake is one of the earliest mobile games that I can remember. I remember playing it many a time, chasing the apple while avoiding contact with the snake's own body. The beauty of the game was that it was a slightly different experience each time you played.

The premise is simple: navigate the snake around the board collecting apples—which increase the length of the snake when consumed—while avoiding collision with the snake itself.
Game update cycle

Before we jump straight into some coding action, let’s first take a look at a couple of core classes that will make our lives easier. When you created your project with the setup tool, the core of the game, the MyGDXGame class, which is the default name of the class, extends a class called ApplicationAdapter. This in turn implements an interface called ApplicationListener. Now, you might think these are good enough for us to get going; however, there is a better class that we can extend and that is the Game class.

What is so special about this class? Essentially, it is ApplicationListener that delegates the game to a screen. Every bar method, such as onCreate(), is implemented. This will save us lots of time going forward.

The following code is the Game class from the LibGDX framework:

```java
public abstract class Game implements ApplicationListener {
    protected Screen screen;

    @Override
    public void dispose () {
        if (screen != null) screen.hide();
    }

    @Override
    public void pause () {
        if (screen != null) screen.pause();
    }

    @Override
    public void resume () {
        if (screen != null) screen.resume();
    }

    @Override
    public void render () {
        if (screen != null) screen.render(Gdx.graphics.getDeltaTime());
    }

    @Override
    public void resize (int width, int height) {
        if (screen != null) screen.resize(width, height);
    }

    public void setScreen (Screen screen) {
```
if (this.screen != null) this.screen.hide();
this.screen = screen;
if (this.screen != null) {
    this.screen.show();
    this.screen.resize(Gdx.graphics.getWidth(), Gdx.graphics.getHeight());
}
}

public Screen getScreen () {
    return screen;
}
}

As we can see in the preceding code, the Game class is abstract, this will require us to provide our own implementation. Then, as mentioned earlier, it delegates life cycle calls to the Screen class.

The Screen class is used to define what a player is looking at, such as the main menu or the game screen. It has various methods that may be overridden. Let's take a look at the following code snippet:

public interface Screen {
    public void show ();
    public void render (float delta);
    public void resize (int width, int height);
    public void pause ();
    public void resume ();
    public void hide ();
    public void dispose ();
}

It has the same method signature as the ApplicationListener class. However, there is an additional method, show(), which is called when the screen becomes the current screen in the game.

We wouldn't want from implement all those methods every time we wanted to create a Screen class implementation. Luckily, LibGDX has an adapter class, ScreenAdapter, which is purely for convenience and contains zero logic. It implements the methods with empty bodies. I won't show the code structure here, I will leave it to you to look it up if you like.

Out of the life cycle methods that are mentioned here, we are only interested in a couple of these to start off with, show() and render().
Let's Get These Snakes Out of This Book!

The render() method is called on every cycle. By default, this is 60 times a second. This is configurable; however, we are happy with 60fps (frames per second) for now. In the default project, you will see that there is some interesting code already. What's happening here is that, with every frame, the screen is being cleared with the background color, which is red in this case, and rendering the LibGDX logo from scratch. This is achieved with the calls to glClearColor() and glClear(). Later, we will look at how we can clear with other colors.

Let's get our Game class up and running. Initially, it will not do much, but it will set us up to make our game.

First, let's generate a new project. This time, we will use some proper names as opposed to the defaults. To create the Snake game, we set the tool as follows:

<table>
<thead>
<tr>
<th>Name:</th>
<th>snake-game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package:</td>
<td>com.pactt.snake</td>
</tr>
<tr>
<td>Game class:</td>
<td>SnakeGame</td>
</tr>
</tbody>
</table>

Once you have the project that is generated from the LibGDX setup application, import it into your IDE; if you are using Eclipse, refer to the previous chapter.

The Game class that is generated for you extends ApplicationAdapter by default.

```java
public class SnakeGame extends ApplicationAdapter {

    public class SnakeGame extends Game {

    Ensure that the imports are updated at the top of the class:

    import com.badlogic.gdx.Game;

    You will now notice that, essentially, nothing has changed. If you run the project again — via DesktopLauncher — it will still show the red screen with the LibGDX logo.

    But why is this happening? We haven't set a Screen class for game object to use. The keen-eyed among you will have spotted that this is because we are still overriding the render() method. Thus, the delegation to the Screen class is not happening.
```
Before we remove the override, let's create our Screen class. Create a class called GameScreen and make it extend by the class ScreenAdapter:

```java
public class GameScreen extends ScreenAdapter {
}
```

As all the interface methods are implemented in the ScreenAdapter class, you will notice that the IDE does not request that you implement anything.

Let's return to our SnakeGame class and remove the default project example code, so all we are left with is the create() method's override:

```java
public class SnakeGame extends Game {
    @Override
    public void create() {
    }
}
```

In the create() method, we can set a new instance of our GameScreen method:

```java
public class SnakeGame extends Game {
    @Override
    public void create() {
        setScreen(new GameScreen());
    }
}
```

If you run the project again, this time you will see that we just have a black screen; however, now the rendering is coming from the screen. Let's draw something on the screen!

**Texture rendering**

Now we have a blank screen, but let's make our screen do something! To start off with, let's put the default project code—the lovely LibGDX logo—onto the screen. You will notice that we don't have a create() method to put the object initialization in. We do, however, have a show() method. Let's put it in there. So now your GameScreen class should look something like this:

```java
public class GameScreen extends ScreenAdapter {
    private SpriteBatch batch;
    private Texture img;

    @Override
```
Let's Get These Snakes Out of This Book!

```java
public void show() {
    batch = new SpriteBatch();
    img = new Texture("badlogic.jpg");
}

@Override
public void render(float delta) {
    Gdx.gl.glClearColor(1, 0, 0, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    batch.begin();
    batch.draw(img, 0, 0);
    batch.end();
}
```

If you run the project again, you will see we are back to where we started. Before we go any further, I am going to provide a little explanation to what is going on, in the code, here.

**The batch class**

Next we need something that will draw our textures; this is where the batch classes come in. A Batch is used to draw 2D rectangles that reference a texture. Among the different Batch class implementations is SpriteBatch, which is the one we will use for our game.

**The texture class**

A texture is a bitmap image that gets drawn on the screen through mapping. The texture class wraps an OpenGL texture without having to worry too much about the internal workings of OpenGL—remember we are here to make games!

When using the texture class, we should ensure that our textures are managed; by this, we mean that if the OpenGL context is lost, for example, that loss can happen by the user switching to another application, then our managed textures will automatically get reloaded for us. Excellent!
Chapter 2

The dispose() method
Everything we have talked about so far, somewhere in the inner workings of LibGDX, holds on to real resources, such as memory. To ensure that our games are well behaved and do not eat all available system memory, we have to remember to dispose of our SpriteBatch and textures when we are done using them. You will notice that they all have a dispose() method in them. When called, this will release all resources associated with that object. The Screen class has a dispose() method. We will look at that later on.

Introducing Sammy the snake
Before we start making the Snake game, we need to set up our textures for the snake and the game play area. So, let's remove the default code from our GameScreen class, leaving just our SpriteBatch batch's clear screen calls:

```java
public class GameScreen extends ScreenAdapter {

  private SpriteBatch batch;

  @Override
  public void show() {
    batch = new SpriteBatch();
  }

  @Override
  public void render(float delta) {
    Gdx.gl.glClearColor(1, 0, 0, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
  }
}
```

Next, let's change the color that fills the screen from red to black. We can do this by updating the glClearColor method call to reference the r, g, b, a values of the black Color class:

```java
Gdx.gl.glClearColor(Color.BLACK.r, Color.BLACK.g, Color.BLACK.b, Color.BLACK.a);
```
Let's Get These Snakes Out of This Book!

If you run the project now, you will find that we are back to our black screen; however, this time, the screen is being cleared every render call. If you don't want to use black, check out the other default colors LibGDX provides, failing that you can define your own! You can do this by creating your own instance of Color with your own r, g, b, a values.

We are very fortunate that there is now plenty of computing power available to do this. Many moons ago, game developers could only refresh sections of the screen that had changed to preserve CPU cycles.

Next we will add back calls in to our batch as we did earlier:

```java
batch.begin();
//Our rendering code will live here!
batch.end();
```

Admittedly they won't do much right now, but our texture rendering code will sit in between them. Failing to call `begin()` before trying to call any other method for the SpriteBatch class will result in a `java.lang.IllegalArgumentException` exception being thrown.

Now, I think we are ready to start drawing our snake!

### Giving the snake a face

We have two assets that we are going to use for drawing our snake, `snakehead.png` and `snakebody.png`. These assets will reside in a specific directory within the project. As we are only using the desktop export of LibGDX at the moment, you will find an `assets` directory in the core project. However, when we start looking at the Android export, you will find the `assets` directory will move. The assets themselves are self-explanatory, one for the snake's head and one for the snake's body. Let's start with the head.

Back in our `GameScreen` class, add a texture object and call it `snakeHead`:

```java
private Texture snakeHead;
```

Next, let's initialize the texture in the `show()` method:

```java
snakeHead = new Texture(Gdx.files.internal("snakehead.png"));
```

Uh oh! Looks like something new cropped up here—`Gdx.files.internal`. LibGDX comes with some handy tools for handling files. Here we are just loading a file from within the project.
Now, let's render it in the `render()` method:

```java
batch.draw(snakeHead, 0, 0);
```

Hopefully, when you run the project, you will get the following output:

![Image of SnakeGame window](image)

The default resolution that the `DesktopLauncher` parameter has is 640 x 480 pixels. Currently, this is what we are rendering to. Later on, we will discuss using a viewport to allow us to handle multiple different resolutions.

**Moving Sammy the snake**

So, we have Sammy the snake on the screen, sitting there, looking snaky. However, it isn't much of a game. If it were, we could finish the book right here! What we need to do now is get that snake slithering across the screen!
First, let's sort out the playing area. Currently, the resolution is 640 x 480 pixels and the snake texture is 32 x 32. This means we have a grid of 20 x 15 — derived by dividing up the resolution by the texture \((640/32 = 20, 480/32 = 15)\) — of the different positions the snake head could be in. The reason we are going to do it this way is because the original game moved with a periodic movement of one snake component at a time. We are going to do the same.

Let's define our timer. We are going to start off with an interval of one second between movements. So let's create a constant field:

```java
private static final float MOVE_TIME = 1F;
```

Now, define a variable to keep track of the time:

```java
private float timer = MOVE_TIME;
```

Finally, let's get it updated in every frame. This is in the `render()` method:

```java
timer -= delta;
if (timer <= 0) {
    timer = MOVE_TIME;
}
```

Here we are deducting the time from the last frame. If the timer reaches zero or below, we reset it.

What we want to do next is move the snake head to the next block once the timer has reached zero or below. Since we only want to make the snake move right, we will be adding 32 px to the position of the snake.

Let's add an \(x\) and \(y\) component for the snake, and a constant for moving:

```java
private static final int SNAKE_MOVEMENT = 32;
private int snakeX = 0, snakeY = 0;
```

Next, let's update the `render()` method:

```java
@Override
public void render(float delta) {
    timer -= delta;
    if (timer <= 0) {
        timer = MOVE_TIME;
        snakeX += SNAKE_MOVEMENT;
    }
    Gdx.gl.glClearColor(Color.BLACK.r, Color.BLACK.g, Color.BLACK.b, Color.BLACK.a);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    batch.begin();
```
batch.draw(snakeHead,snakeX,snakeY);
batch.end();
}

So now every time the timer reaches zero or below, the snake will move to the right by 32 px.

Run the default project and check it out!

So what did you see? Hopefully, it was the snake head scrolling across the screen in intervals of 1 second. Then it probably disappeared off to the right? Yes, we should have been expecting that. We told the snake head to move to the right indefinitely. What we should do, staying true to the original, is have the snake reappear on the left-hand side.

Let's create a method to check for the position of the snake:

```java
private void checkForOutOfBounds() {
    if (snakeX >= Gdx.graphics.getWidth()) {
        snakeX = 0;
    }
}
```

Now we should have this method called after we move the snake. What we are doing here is checking whether the position of the snake head is outside the screen. We do this by querying the screen's width and checking whether the snake's x position is greater than or equal to it. We then reset it to zero.

Running the project now, you will see that the snake head will return back to the start of the screen.

This is all well and good, but again, it isn't much of a game just having the snake move to the right. What we need to do is give the snake a direction:

```java
private static final int RIGHT = 0;
private static final int LEFT = 1;
private static final int UP = 2;
private static final int DOWN = 3;
private int snakeDirection = RIGHT;
```

By adding some constants and a variable, we can keep track of the snake's direction. Now that we have a direction, we need to be more precise on how we mean to move the snake. Let's replace the current code for moving with the following method:

```java
private void moveSnake() {
    switch (snakeDirection) {
        case RIGHT: {
```
Let's Get These Snakes Out of This Book!

```java
snakeX += SNAKE_MOVEMENT;
return;
}
case LEFT: {
    snakeX -= SNAKE_MOVEMENT;
    return;
}
case UP: {
    snakeY += SNAKE_MOVEMENT;
    return;
}
case DOWN: {
    snakeY -= SNAKE_MOVEMENT;
    return;
}
```

Now, depending the direction set, the snake will move; give it a go and set the snake to move in a different direction.

Ah, I bet you've seen the problem with this already. We only check whether the snake is on the screen when moving right, and that means we need to update our bounds to check method to cover all cases:

```java
private void checkForOutOfBounds() {
    if (snakeX >= Gdx.graphics.getWidth()) {
        snakeX = 0;
    }
    if (snakeX < 0) {
        snakeX = Gdx.graphics.getWidth() - SNAKE_MOVEMENT;
    }
    if (snakeY >= Gdx.graphics.getHeight()) {
        snakeY = 0;
    }
    if (snakeY < 0) {
        snakeY = Gdx.graphics.getHeight() - SNAKE_MOVEMENT;
    }
}
```

That should be better!

Just before we move on to controlling the snake, let's do a little code refactor to stop things from getting a bit too messy in our `render()` method. You will see in the samples that this has already been done!
Chapter 2

Controlling Sammy with event polling

Now we are going to look at how to control our snake. Normally, there are a couple of approaches to handle user input. One is event polling, where we can query the state of the input and then update our game accordingly. Generally, this polling will occur during the main game loop, just before the rendering. The other method is to listen to events; we register a listener that will receive events from the LibGDX framework when the player interacts with the game, that is, presses a key. For now, we will just work with polling.

As we are working in desktop mode, we will take advantage of the arrow keys on the keyboard. Because LibGDX is so awesome, there is a really convenient way to poll for user input, Gdx.input. There is a vast array of methods that we can call here to determine what the player has pressed. For this example, we will focus on the Gdx.input.isKeyPressed() method. Here we can specify the key to query by passing in a key code. The codes are available as constants that live inside the Keys static class; please check the official documentation at http://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/Input.Keys.html to access the whole list.

So how do we use it?

Well, let's create a method called queryInput() in it. We will check whether the arrow keys have been pressed and update the snake direction accordingly:

```java
private void queryInput() {
    boolean lPressed = Gdx.input.isKeyPressed(Input.Keys.LEFT);
    boolean rPressed = Gdx.input.isKeyPressed(Input.Keys.RIGHT);
    boolean uPressed = Gdx.input.isKeyPressed(Input.Keys.UP);
    boolean dPressed = Gdx.input.isKeyPressed(Input.Keys.DOWN);

    if (lPressed) snakeDirection = LEFT;
    if (rPressed) snakeDirection = RIGHT;
    if (uPressed) snakeDirection = UP;
    if (dPressed) snakeDirection = DOWN;
}
```

So firstly, we query the four keys we are interested in, and then we set the direction.

Now, add this method to the top of the render() method.

Run the project and you will find that you can control the snake with the arrow keys.
Adding the apple
Next up, we need to get our snake eating something. Let's get our apple implemented.

We will need to get our apple to do the following things:

- Randomly being placed on the screen, not on the snake!
- Only place one if there isn't an apple on the screen already
- Disappear when it collides with the snake's head

Right! Let's add the texture:

```java
private Texture apple;
```

Then, let's amend our `show()` method and add the code to instantiate the apple texture:

```java
apple = new Texture(Gdx.files.internal("apple.png"));
```

Let's add a flag to determine if the apple is available:

```java
private boolean appleAvailable = false;
private int appleX, appleY;
```

This will control whether or not we want to place one. In the Snake game, the next apple appears after the current apple is eaten; therefore, we don't need any fancy timing mechanism to deal with it. Hence, the apple is not available at the start as we need to place one first. We also specify the variables that will contain the location of the apple.

Finally, add the drawing code to the `render()` method:

```java
if (appleAvailable) {
    batch.draw(apple, appleX, appleY);
}
```

Here, we tell the game to only render the apple if it has been placed.

Now we need to randomly place the apple on the screen:

```java
private void checkAndPlaceApple() {
    if (!appleAvailable) {
        do {
            appleX = MathUtils.random(Gdx.graphics.getWidth() / SNAKE_MOVEMENT - 1) * SNAKE_MOVEMENT;
            appleY = MathUtils.random(Gdx.graphics.getHeight() / SNAKE_MOVEMENT - 1) * SNAKE_MOVEMENT;
        } while (appleAvailable);
    }
}
```
appleAvailable = true;
} while (appleX == snakeX && appleY == snakeY);
}

The previous code listing shows the required rule for placing the apple on the
screen. First, we check whether we need to place an apple, then we randomly pick
a location on the screen, which is a multiple of 32, and we repick it if the picked
location contains the snake. As we are working in a 0-indexed environment we need
to subtract one (1-20 becomes 0-19 and 1-15 becomes 0-14). Add this method to the
render() method after we move the snake:

```java
@Override
public void render(float delta) {
    queryInput();
    timer -= delta;
    if (timer <= 0) {
        timer = MOVE_TIME;
        moveSnake();
        checkForOutOfBounds();
    }
    checkAndPlaceApple();
clearScreen();
draw();
}
```

You may have noticed two new methods creep in there, clearScreen() and draw().
These are the result of a very quick refactor to neaten up our render method. The
clearScreen() method is as follows:

```java
private void clearScreen() {
    Gdx.gl.glClearColor(Color.BLACK.r, Color.BLACK.g,
    Color.BLACK.b, Color.BLACK.a);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
}
```

The draw() method is as follows:

```java
private void draw() {
    batch.begin();
    batch.draw(snakeHead, snakeX, snakeY);
    if (appleAvailable) {
        batch.draw(apple, appleX, appleY);
    }
    batch.end();
}
```
Now run the project. Hopefully you will see something similar to the following screenshot. The apple will be randomly placed on the screen every time you launch the project.

You may have noticed, however, that if you take control of the snake and head over to the apple, the snake will just glide over the top and the apple will remain. Not very game-like and not what we would want to happen.

Let's fix this!

We can do this quite simply by checking whether the apple and the snake head coordinates match:

```java
private void checkAppleCollision() {
    if (appleAvailable && appleX == snakeX && appleY == snakeY) {
        appleAvailable = false;
    }
}
```
If they do collide, we set the appleAvailable flag to false. This will trigger a respawn of the apple.

Add the checkAppleCollision() method to the render() method above checkAndPlaceApple().

Run the project, and you will now find that the apple respawns every time the snake head collides with it.

On to the final part of this chapter. We need to make the snake grow in length every time it eats an apple.

**Increasing the length of the snake**

We have the snake eating the apple; however, there aren't any consequences to this. As a part of the game, we need to make the snake increase the length of his body for each apple he eats. Our requirements for this are as follows:

- Add a body part to the snake when it eats an apple
- As the snake moves, the body parts should follow
- There will be multiple body parts

First, let's create a class that will contain the length of the snake's body. This can take the form of an inner class for now:

```java
private class BodyPart {
    private int x, y;
    private Texture texture;

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

    public void updateBodyPosition(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void draw(Batch batch) {
        if (!(x == snakeX && y == snakeY)) batch.draw(texture, x, y);
    }
}
```
As you can see, it contains the x and y components along with the texture to be drawn. It also has a `draw()` method which will only draw the texture if the body part is not in the same location as the snake's head. This is important for when you collect the apple; you wouldn't want the body part showing over the head before the snake moves.

Next, let’s create the texture:

```java
private Texture snakeBody;
```

Again in our `show()` method, let’s add code to instantiate the object:

```java
snakeBody = new Texture(Gdx.files.internal("snakeBody.png"));
```

Let’s create an array of body parts:

```java
private Array<BodyPart> bodyParts = new Array<BodyPart>();
```

Here, we are using LibGDX’s built-in `Array<T>` class. It has some useful methods for accessing the first and last element of the array that we will need shortly.

Now, when that part is set up, we can add a body part when a collision between the snake’s head and apple occurs:

```java
private void checkAppleCollision() {
    if (appleAvailable && appleX == snakeX && appleY == snakeY) {
        BodyPart bodyPart = new BodyPart(snakeBody);
        bodyPart.updateBodyPosition(snakeX, snakeY);
        bodyParts.insert(0, bodyPart);
        appleAvailable = false;
    }
}
```

When a collision is detected, we create a new body part, set its position, and insert it at the front of the array. We are reusing the texture for the body parts, so we can reduce our memory footprint by not loading it multiple times. We can do this as the drawing of the texture doesn't require it to be a different instance of the texture every time we draw it in the same render cycle.

Next, we need to update the body parts every time the snake moves; if we don’t do this, they will remain in their place for all eternity. Let’s add the following code to our game:

```java
private int snakeXBeforeUpdate = 0, snakeYBeforeUpdate = 0;
private void moveSnake() {
    snakeXBeforeUpdate = snakeX;
    snakeYBeforeUpdate = snakeY;
```
private void updateBodyPartsPosition() {
    if (bodyParts.size > 0) {
        BodyPart bodyPart = bodyParts.removeIndex(0);
        bodyPart.updateBodyPosition(snakeXBeforeUpdate,
                                  snakeYBeforeUpdate);
        bodyParts.add(bodyPart);
    }
}

Right, first we need to keep track of the previous position of the snake—the reason for this will be clear in a minute. Then we have the updateBodyPartsPosition() method, where we take the front element of the array, which would be the body part that is the tail part of the snake, and we remove it from the array. We then update the position and add it to the back of the array, which is the front of the snake. This means we are only updating the one body part as the others won’t need to move unless they are the tail of the snake.

Add a call to this method after we check whether the snake is out of bounds:

    // render method
    moveSnake();
    checkForOutOfBounds();
    updateBodyPartsPosition();

Before we run the project and see what we have created, we need to update our draw() method to draw out the body of the snake:

    private void draw() {
        batch.begin();
        batch.draw(snakeHead, snakeX, snakeY);
        for (BodyPart bodyPart : bodyParts) {
            bodyPart.draw(batch);
        }
        if (appleAvailable) {
            batch.draw(apple, appleX, appleY);
        }
        batch.end();
    }
With that in place, if you run the project now, you will see that the snake grows every time it eats an apple.

Summary

In this chapter, we looked at setting up our game to use the screen-based system, we then implemented our own game using the `Game` class. We looked at the render cycle and the loaded textures and handled input. We have the makings of a very good game!

Coming up, we will take this game further and make it complete. We will do this by adding the collision of the snake with itself, creating a gameover scenario, and introducing a scoring system.
Where to buy this book

You can buy LibGDX Game Development By Example 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.