LibGDX Cross-Platform Development Blueprints

LibGDX is a very popular open source game framework for the Java programming language.

LibGDX Cross-Platform Development Blueprints teaches you the concepts of game development using the LibGDX framework as you make four complete games.

You’ll start by setting up the environment, then move on to advanced concepts such as collision detection, memory optimization, and more. The first game is Monty Hall, where you’ll learn how to set up LibGDX and use simple graphics. Then, you’ll get to know more about concepts such as animation, game sounds, and scoring by developing a Whack-A-Mole game. This will set up the base for Catch the Ball game, where you’ll get to grips with concepts such as movement and collisions. Finally, the Dungeon Bob game will help you understand platformer game concepts.

This guide gives you everything you need to master game development with LibGDX.

Who this book is written for

If you have a good grip on Java and want to explore its capabilities in game development, this book is for you. Basic knowledge of LibGDX is preferred, but is not mandatory.

What you will learn from this book

- Set up the development environment and implement a very simple game type
- Add features such as motion, sounds, and randomness while creating a physics based game
- Add music, physics, and menus to your games
- Start the creation of a platformer game and apply optimization techniques
- Perform collision detection and manage game assets
- Learn about a tool used for making game levels and its features
- Create enemies, multiple levels, and level transitions in the game

LibGDX Cross-Platform Development Blueprints

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

Indraneel Potnis

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Catch the Ball'
- A synopsis of the book’s content
- More information on LibGDX Cross Platform Development Blueprints
Indraneeel Potnis is a mobile developer who lives in Mumbai. He has worked in diverse areas of the IT industry, such as web development, QA, and mobile application development.

Since childhood, he has been interested in playing computer games, and he became interested in making them in college. He made a card game called Mendhicoat with a friend on the Android platform and released it on the Google Play store.
LibGDX is a game framework with which people can make efficient games that run on all the platforms (mobile/web/desktop) with a single code base. Games are also a big source of monetization in the mobile market. The programming language is Java, which is widely used everywhere, and is very easy to learn.

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

What this book covers

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

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

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

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

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

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

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

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

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

Chapter 10, More Levels and Effects, discusses how to make multiple levels, a loading screen, and particle effects, among other things.
In this chapter, we will learn how to make a game called Catch the Ball. The user has to catch a ball thrown from a height in a basket. The ball will be randomly thrown from above. The user would be given a point from where he needs to catch the ball. We will display the score and also the highest score for the game.

The following topics will be covered in this chapter:

- Making a moving basket
- Throwing the ball
- Detecting collisions
- Throwing multiple balls
- Keeping score and saving the high score
- Implementing screens
- Adding sound effects and music
Making a moving basket
Set up a project similar to the one I have, as shown here:

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

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

Type the following code in the file:

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

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

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

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

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

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

Implementing the GameManager class

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

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

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

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

private static float BASKET_RESIZE_FACTOR = 3000f;

public static void initialize(float width,float height){
    basket = new Basket();
    basketTexture = new Texture(Gdx.files.internal
    ("data/basket.png"));
    basket.basketSprite = new Sprite(basketTexture);
    basket.basketSprite.setSize(basket.basketSprite.
    getWidth()*(width/BASKET_RESIZE_FACTOR), basket.
    basketSprite.getHeight()*(width/BASKET_RESIZE_FACTOR));
    // set the position of the basket to bottom - left corner
    basket.setPosition(0, 0);
    backgroundTexture = new Texture(Gdx.files.internal
    ("data/background.jpg"));
    backgroundSprite= new Sprite(backgroundTexture);
    // set the background to completely fill the screen
    backgroundSprite.setSize(width, height);
}

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

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

Implementing the CatchTheBall class

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

package com.packtpub.catchtheball;

import com.badlogic.gdx.ApplicationAdapter;

[68]
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.packtpub.catchtheball.managers.GameManager;

public class CatchTheBall extends ApplicationAdapter {
    SpriteBatch batch; // spritebatch for drawing
    OrthographicCamera camera;

    @Override
    public void create () {
        // get window dimensions and set our viewport dimensions
        float height= Gdx.graphics.getHeight();
        float width = Gdx.graphics.getWidth();
        // set our camera viewport to window dimensions
        camera = new OrthographicCamera(width,height);
        // center the camera at w/2,h/2
        camera.setToOrtho(false);
        batch = new SpriteBatch();
        //initialize the game
        GameManager.initialize(width, height);
    }

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

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

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

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

batch.end();

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

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

```java
public void handleTouch(float x, float y){
    if(x-(basketSprite.getWidth()/2)>0.0){
        setPosition(x-(basketSprite.getWidth()/2), 0);
    } else{
        setPosition(0,0);
    }
}
```
This method will set the basket's $x$ coordinate to wherever the user has touched/clicked on the screen. We will set the position in such a way that the basket's center coincides with the touch coordinate. But if the user touches too close to the left end of the screen, the basket will be drawn outside the visible area. In that case, we just set the basket's position to $(0, 0)$.

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

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

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

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

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

public class InputManager extends InputAdapter {
    OrthographicCamera camera;
    static Vector3 temp = new Vector3();

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

    @Override
    public boolean touchUp(int screenX, int screenY, int pointer, int button) {
        temp.set(screenX, screenY, 0);
        // get the touch coordinates with respect to the camera's viewport
        camera.unproject(temp);
    }
}
```
float touchX = temp.x;
float touchY = temp.y;

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

}
Chapter 3

Throwing the ball

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

Making the ball

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

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

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

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

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

```java
static Ball ball; // ball instance
static Texture ballTexture; // texture image for the ball
private static final float BALL_RESIZE_FACTOR = 2500f;
```

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

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

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

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

```java
ball.render(batch);
```
We will dispose of the texture in the dispose() method:

    ballTexture.dispose();

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

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

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

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

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

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

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

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

    ball.update();
    ball.render(batch);
Now if you run the game, you should see the ball falling from above.

**Adding gravity**

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

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

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

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

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

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

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

**Detecting collisions**

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

**Colliding with the ground**

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

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

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

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

Colliding with the basket

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

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

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

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

```java
ball.velocity.set(0, 0);
Vector2 center = new Vector2();
// set the center at the center of ball sprite
center.x=ball.position.x + (ball.ballSprite.getWidth()/2);
center.y=ball.position.y + (ball.ballSprite.getHeight()/2);
ball.ballCircle = new Circle(center, (ball.ballSprite.getHeight()/2));
```

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

```java
ballSprite.setPosition(position.x, position.y);
    // set the position of the sprite
ballCircle.setPosition(position.x+ (ballSprite.getWidth()/2),
    (position.y+ ballSprite.getHeight()/2));
```
We will follow similar steps for the basket. In the `Basket` class, add the following line of code:

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

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

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

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

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

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

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

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

```java
public boolean checkCollisionsWithBasket(){
    // check if the ball collided with the basket
    if(Intersector.overlaps(ballCircle, GameManager.basket.basketRectangle)){
        System.out.println("Collided with basket");
        return true;
    }
    return false;
}
```
LibGDX has a utility class called Intersector to detect intersections between different shapes. We use its overlaps() method to check for collisions between a circle and a rectangle. We will call these two functions in the new checkCollisions() method:

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

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

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

Let's take a look at the following diagram:

![Diagram of a ball and a rectangle]

## Throwing multiple balls

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

### Throwing the balls after specific intervals

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

```java
public boolean isAlive; // flag to indicate if the ball is alive or not
```
We will set the flag to false if it collides with either the basket or the ground. In the checkCollisionsWithBasket() method, add the following lines of code:

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

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

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

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

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

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

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

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

    public static Array<Ball> balls = new Array<Ball>();
    // array of ball objects
We will create a new class called \texttt{SpawnManager} that handles the creation and deletion of new \texttt{Ball} objects based on the interval:

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

import com.badlogic.gdx.graphics.Texture;

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

    static float width, height; // viewport width and height
    static Texture ballTexture; // texture image for the ball

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

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

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

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

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

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

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

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

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

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

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

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

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

    SpawnManager.initialize(width, height, ballTexture);

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

    SpawnManager.run(balls);
    for(Ball ball:balls){
        if(ball.isAlive){
            ball.update();
            ball.render(batch);
        }
    }
    SpawnManager.cleanup(balls);

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

**Randomizing and optimizing**

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

    static Random random = new Random(); // object of random class to generate random numbers
In our `createNewBall()` method, we set the x coordinate to 0 for the ball. Replace this line with the following:

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

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

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

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

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

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

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

```java
public static Ball resetBall(Ball ball){
    ball.ballSprite.setSize(ball.ballSprite.getWidth()*width/BALL_RESIZE_FACTOR,ball.ballSprite.getHeight()*width/BALL_RESIZE_FACTOR);
    ball.position.set(random.nextInt((int) (width - ball.ballSprite.getWidth())), height-ball.ballSprite.getHeight());
    ball.velocity.set(0, 0);
    ball.isAlive=true;
    Vector2 center = new Vector2();
    //set the center at the center of ball sprite
    center.x=ball.position.x + (ball.ballSprite.getWidth()/2);
    center.y=ball.position.y + (ball.ballSprite.getHeight()/2);
    ball.ballCircle = new Circle(center, (ball.ballSprite.getHeight()/2));
    return ball;
}
```

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

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

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

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

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

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

**Keeping the score and maintaining the high score**

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

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

    public static int score;

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

    score=0;

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

    package com.packtpub.catchtheball.managers;

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

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

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

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

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

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

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

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

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

Finally, in the `Ball` class' `checkCollisionsWithBasket()` method, we increase the score when we catch the ball with the basket:

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

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

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

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

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

In the `Font` section, there is a file input area where you can select the TTF file. Once you select it, you can see how the font looks in the rendering section. To keep it simple, we will not add any extra effects:
Save the font by navigating to File | Save BMFont files (text). Give the file a .fnt extension and save it. Hiero creates one more file with the .png extension. You can actually open the image in any image viewer/editor to see how the font characters are stored. To load the font in our game, create a new folder called fonts in the assets/data directory. Copy the font file and the image to this folder.

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

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

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

```java
font.setScale(width/1400f);
```
That's it. You can now see the score text in your custom font:

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

**Saving high scores**

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

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

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

Next, let's declare the variable for preferences:

```java
static Preferences prefs; // preferences instance
```
In the `initialize()` method, add the following two lines:

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

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

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

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

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

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

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

This is similar to what we did for the score, except that here we will display the high score text in the top-left corner of our screen.
Implementing screens
In this section, we will learn how to implement a menu screen for our game and how to transition between it and the game screen.

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

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

```java
package com.packtpub.catchtheball;

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

    @Override
    public void show() {
    }

    @Override
    public void render(float delta) {
    }

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

    @Override
    public void pause() {
    }

    @Override
    public void resume() {
    }
```
To implement a screen in LibGDX, we have to implement the `Screen` interface. As it is an interface, we will have to implement all the methods from it. These methods are similar to the `ApplicationListener` interface, which we saw earlier. It adds the two `show()` and `hide()` methods. These methods are called when the screen is being shown (active) and when the screen is hidden (deactivated).

Let's declare some variables in this class:

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

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

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

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

```java
public MenuScreen(){
    // get window dimensions and set our viewport dimensions
    float height= Gdx.graphics.getHeight();
    float width = Gdx.graphics.getWidth();
```
// set our camera viewport to window dimensions
camera = new OrthographicCamera(width, height);

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

batch = new SpriteBatch();

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

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

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

// set the size and positions
startButtonSprite.setSize(startButtonSprite.getWidth()*(width/BUTTON_RESIZE_FACTOR), startButtonSprite.getHeight()*(width/BUTTON_RESIZE_FACTOR));
exitButtonSprite.setSize(exitButtonSprite.getWidth()*(width/BUTTON_RESIZE_FACTOR), exitButtonSprite.getHeight()*(width/BUTTON_RESIZE_FACTOR));
backGroundSprite.setSize(width, height);

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

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

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

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

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

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

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

```java
startButtonTexture.dispose();
extButtonTexture.dispose();
batch.dispose();
```

### Implementing screen transitions

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

```java
package com.packtpub.catchtheball;
import com.badlogic.gdx.Game;

public class MainGame extends Game {
    @Override
    public void create() {
        setScreen(new MenuScreen());
    }
}
```
In the `create()` method, we call the `setScreen()` method to change our currently displayed screen to `MenuScreen` by passing an instance of it. In the launcher classes, pass the instance of this class. For example, in the desktop launcher, it is implemented as follows:

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

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

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

```java
public class CatchTheBall implements Screen {

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

```java
public void render(float delta) {
```
In the MenuScreen class, let's declare an instance of the MainGame class. We will need this to call its setScreen() method to transition between the screens. Parameterize the constructor of the MenuScreen class to set this instance:

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

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

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

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

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

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

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

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

        // handle touch input on the start button
        if ((touchX >= startButtonSprite.getX()) && touchX <= (startButtonSprite.getX() + startButtonSprite.getWidth()) && (touchY >= startButtonSprite.getY()) && touchY <= (startButtonSprite.getY() + startButtonSprite.getHeight())) {
            game.setScreen(new CatchTheBall()); // Bring the game screen to front
        }
    }
}
```
In this method, after capturing the input coordinates, we first check whether the user has touched the Start button. If he has touched it, then we bring the game screen to the front by calling the setScreen() method of the game. If the user has touched the Exit button, then we quit the application. We call this method in the render() method:

    batch.end();

    handleTouch();

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

    public void hide() {
        dispose();
    }

### Implementing the Back button

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

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

    CatchTheBall (MainGame game) {
        CatchTheBall.game = game;
    }
We will pass the reference from the MenuScreen class:

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

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

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

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

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

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

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

Render it in the `renderGame()` method:

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

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

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

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

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

[99]
backButtonSprite.getY()) && touchY<=(GameManager.
backButtonSprite.getY()+GameManager.
backButtonSprite.getHeight()) ){
    CatchTheBall.game.setScreen(new MenuScreen
    (CatchTheBall.game)); // Bring the menu screen to front
}
}

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

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

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

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

The screen will now look like this:
Catching the Back button

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

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

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

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

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

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

Let's take a look at the following diagram:
Adding sound effects and music
In this section, we will add collision sound effects and background music to our game.

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

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

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

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

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

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

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

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

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

```java
//dispose the sound instances
groundHitSound.dispose();
basketHitSound.dispose();
```
Adding background music

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

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

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

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

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

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

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

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

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

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

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

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

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

To pause the music, we can call the pause() method on the instance, and to stop it, there is a stop() method as well.
Summary
In this chapter, we made a game called Catch the Ball and learned some concepts along the way. These include the following:

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

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


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

[Click here](http://www.PacktPub.com) for ordering and shipping details.