ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Desenvolvimento de Aplicações Móveis
Mobile Application Development
DAM

Tutorial 4 - LibGDX

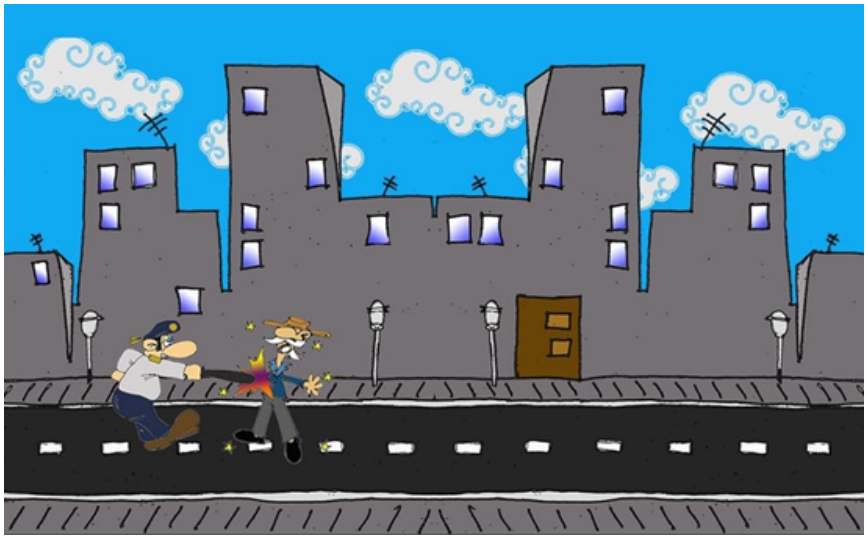
António Teófilo
ADEETC - C.2.4
antonio.teofilo@isel.pt

Pedro Fazenda
CEDET - E.2.16.4
pedro.fazenda@isel.pt

Abstract

This tutorial aims to develop a Game using LibGDX: Textures, Animations, Accelerometer, Events, Background Scrolling Parallax and Audio.

Deadline: 31 May 2020



LibGDX

11 May 2020

Contents

List of Figures

1 Introduction

This work aims to introduce a set of techniques commonly used in the development of games for Android using the game engine libGDX. The application developed in this tutorial includes textures, animations with Sprites, the use of the accelerometer, events, Background Parallax and audio. Follows a set of links that you should consult during the development of this work:

- Android Developers: `http://developer.android.com/training/index.html`
- libGDX Site: `http://libgdx.badlogicgames.com/index.html`
- libGDX Wiki: `https://github.com/libgdx/libgdx/wiki`
- Box2D Manual: `https://github.com/libgdx/libgdx/wiki/Box2d`

This work is to be done in **Java**, and its 10% for the Kotlin implementation is included in the Java implementation.

2 Project P1Planes - Hello libGDX, textures and animations

The application developed in this project includes animations with Sprites, the use of the accelerometer and events to capture the touch on the screen.

Figure 1: Screen of Project P1Planes.

2.1 libGDX Project

1. Download the *gdx-setup.jar* file (from libGDX Site select Download Setup app) to generate a libGDX project for Android. This file is an executable Jar file that launch a user interface to configure and generate a libGDX project. First we will generate a libGDX project with all the libGDX resources and then we will open it in Android Studio to add code and resources. Run the file (*gdx-setup.jar*). To run on the command line do: *java -jar gdx-setup.jar*

2. In the user interface (Figure 2) specify the project/application name as P1Planes, the package name, the name of the main class, the directory (p2Faces) where the project will be created (choose a directory in a dedicated directory for libGDX projects) and indicate the location of the Android SDK. Next, for this project, select only Android as the platform on which the project will run. Finally, select the extension "Box2d" (the only one we will use in this tutorial) and generate the project. If you have a more recent Android Build tools, select to use it.

Figure 2: libGDX P1Planes Project Setup.

3. Open the project in Android Studio (AS) (Open, and select previous directory). Check that the project consists of 3 parts (at top level): **android**, **core** and gradle. The android part consists of the usual components in an Android project. The gradle part has "build.gradle" files for: the Android part; the core part; and the whole project. We will develop our application in the **core** part. Below is the generated code that creates and loads our libGDX application within the Android project. Run the program.

```
public class AndroidLauncher extends AndroidApplication {
    @Override
    protected void onCreate (Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        AndroidApplicationConfiguration config = new AndroidApplicationConfiguration();
        initialize(new MyGdxGame(), config);
    }
}
```

Listing 1: Planets Resource Arrays.

2.2 libGDX Framework

1. A libGDX application, in Android, has the life-cycle quite similar to an Android application. This life-cycle is described in Figure 3. Our game class (*MyGdxGame* by default) can receive these life-cycle events, as it implements the *ApplicationListener* interface. This interface defines callback methods for the events: create, dispose, pause, resume, resize and render. In fact, our application class extends from an adapter class (*ApplicationAdapter*), which provides empty implementations for all those methods, and that is why we, in our class, do not need to declare all those callbacks. Examine: *MyGdxGame* class, *ApplicationAdapter* and *ApplicationListener* (in AS give a CTRL+click on their names).

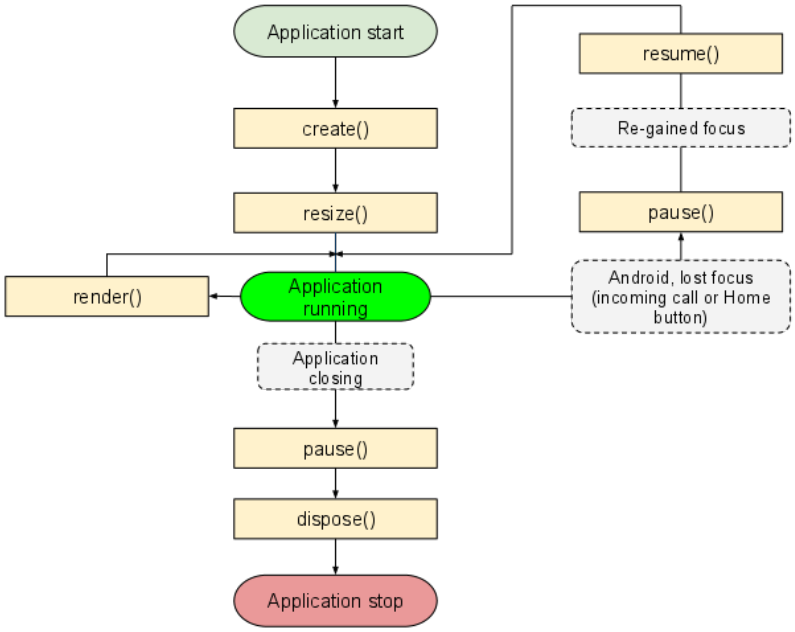


Figure 3: libGDX App Life-cycle.

```
public interface ApplicationListener {
   // Called when the app is first created.
   public void create() { ... }

   // Called when the app should render itself
   public void render() { ... }

   // Called when the app is resized or after create(). This can happen at any
   // point during a non-paused state but will never happen before a call to
   // create()
   public void resize(int width, int height) { ... }

   // Called when the app is paused, usually when it's not active or visible
   // on-screen. An Application is also paused before it is destroyed.
   // A good place to save the game state.
   public void pause() { ... }

   // Called when the app is resumed from a paused state, usually when it
   // regains focus
   public void resume() { ...}

   // Called when the app is destroyed. Preceded by a call to pause()
   public void dispose() { ... }
```

```
}
```

Listing 2: ApplicationListener interface.

2. LibGDX consists of 6 main interfaces (Application, Files, Input, Net, Audio and Graphics) define the way for the game class to interact with the operating system. These 5 interfaces are materialized in five objects, with which the application may interact to use the respective functionality: Gdx.app, Gdx.files, Gdx.input, Gdx.net, Gdx.audio and Gdx.gl.

   - **Application**: [class web link] The Application class executes the application and notifies it of the events that occur during its execution. It provides "login" facilities and methods for questioning the system state, e.g., memory usage.

```
// Android Version
int androidVersion = Gdx.app.getVersion();

// memory consumption
long javaHeap = Gdx.app.getJavaHeap();
long nativeHeap = Gdx.app.getNativeHeap();
```

Listing 3: Some Application functionality: Get Version And Memory.

   - **Files**: [class web link] The Files class allows access to the platform's file system. It provides an abstraction about the different file locations.

```
// Image file in the assets directory
String fileName = "assets/texture/brick.png";
Texture myTexture = new Texture(Gdx.files.internal(fileName));
```

Listing 4: Accesssing the file system.

   - **Input**: [class web link] The Input class notifies the application of user input events related to: mouse, keyboard, touch screen, accelerometer, ...

```
// current touch coordinates if a touch event is in progress
if (Gdx.input.isTouched()) {
   System.out.println("Input occurred at x=" + Gdx.input.getX()
                      + ", y=" + Gdx.input.getY());
}
```

Listing 5: Getting Touch Coordinates.

   - **Net**: [class web link] The Net class enables to perform networking operations, such as simple HTTP get and post requests, and TCP server/client socket communication.
   - **Audio**: [class web link] The audio class enables the reproduction of sounds, sound effects and streaming music and to access audio input/output devices.

```
// plays a sound file from disk repeatedly with the volume half turned up
String fileName = "data/myMusicFile.mp3";
Music = Gdx.audio.newMusic(
              Gdx.files.getFileHandle(fileName, Files.FileType.Internal));
music.setVolume(0.5f);
music.play();
```

Listing 6: Playing Sound.

- **Graphics**: [class web link] The Graphics class provides access OpenGL ES 2.0/3.0 and enables to query video configuration parameters and other similar media.

```
// clears the screen and paints it with red
Gdx.gl.glClearColor(1, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
```

Listing 7: Clear And Paint Red.

Give a look to these interfaces, using the web links or, in code, check the contents of Gdx class (CTRL+click on it) and (inside it) the contents of the interface types that support the references: app, files, ...

3. **Logging**. The Application interface has three methods to log debug, informative and error messages, as shown in Listing 8. The log level can be controlled by the *Gdx.app.setLogLevel* method: the log level defines which log messages are shown - it enables to log messages with their level and use the *setLogLevel* to define the level of messages that we want to see. The use of informative messages and LOG_INFO level should be the default. Log a message on *enter*, *render*, *dispose* methods on your app, with text *onEnter*, *onRender* and *onDispose*. Add the pause and resume methods and add the log message to them. Watch the log messages in LogCat. Comment the message in render - too many messages. From now on, use the logging to show app state or make debug.

```
int logLevel = Application.LOG_DEBUG;
Gdx.app.setLogLevel(logLevel);
// Application.LOG_NONE: mutes all logging.
// Application.LOG_DEBUG: logs all messages (DIE).
// Application.LOG_INFO: logs informative and error messages (IE).
// Application.LOG_ERROR: logs only error messages (E).

Gdx.app.debug("MyTag", "A debug message");
Gdx.app.log("MyTag", "An informative message");
Gdx.app.error("MyTag", "An error message");
```

Listing 8: Logging.

2.3 Hello libGDX

1. **Change background colour and center image**. Change the background colour and place the image at the center of the screen (see code). Here and after any changes: run the program, check for the result, change the code and repeat all these steps, until you get the desired effect.

```
// in render method
batch.begin();
batch.draw(img, Gdx.graphics.getWidth() / 2f - img.getWidth() / 2f,
                Gdx.graphics.getHeight() / 2f - img.getHeight() / 2f);
batch.end();
```

Listing 9: Render code for an image.

2. **Add some text**. To place text we need a *BitmapFont* object. We may create it in *create* method and dispose it (font.dispose()) in *dispose* method. The code for

the *create* and *render* methods is in the Listing 10. When adding code from these Listing, add the necessary imports and class fields (use the AS quick actions for that (Alt+Enter)).

```
// put on create mtehod
font = new BitmapFont();
font.setColor(Color.RED);
font.getData().setScale(4.0f, 4.0f);

// add to render method
batch.begin();
...
font.draw(batch, "Hello libGDX - Tutorial 4", 50, screenHeight - 50);
batch.end();
```

Listing 10: Code for text.

3. **Import resources**. Copy the following files to the *android/assets* directory of the project: *plain.png*, *spritesheet.png* and *spritesheet.atlas*.

4. **Set new picture**. Replace the existing picture with the image *plain.png* from provided resources.

2.4 Dynamic Textures

1. **Create a texture**. Textures can be create dynamically using the object *Pixmap*. Add the lines of code in Listing 11 to *create* and *render* methods, which will create two red areas in the screen.

```
// To create method
// create 256 wide, 128 height using 8 bits for Red, Green, Blue and Alpha
pixmap = new Pixmap(256, 128, Pixmap.Format.RGBA8888);
pixmap.setColor(Color.RED);
pixmap.fill();

// Draw two lines forming an X
pixmap.setColor(Color.BLACK);
pixmap.drawLine(0, 0, pixmap.getWidth() - 1, pixmap.getHeight() - 1);
pixmap.drawLine(0, pixmap.getHeight() - 1, pixmap.getWidth() - 1, 0);

// Draw a circle in the middle
pixmap.setColor(Color.YELLOW);
pixmap.drawCircle(pixmap.getWidth() / 2, pixmap.getHeight() / 2,
                                         pixmap.getHeight() / 2 - 1);

// create a sprite, that is a texture
sprite = new Sprite(new Texture(pixmap));
pixmap.dispose();

// Add to render method, between batch.begin() and batch.end()
sprite.setPosition(0, Gdx.graphics.getHeight() / 2f - sprite.getHeight()/2);
sprite.draw(batch);
sprite.setPosition(Gdx.graphics.getWidth() - sprite.getWidth(),
                   Gdx.graphics.getHeight() / 2f - sprite.getHeight() / 2);
sprite.draw(batch);
```

Listing 11: Code for lines, areas and Textures.

2.5 Animations

1. **Build an animation**. We will use the *spritesheet.png* and the *spritesheet.atlas* files, where the first one contains several images into a single image, and the second one is a text file with a kind of map (atlas) with the location of the images in the first one. Examine the contents of both files. Add the code below for the *create* and *render* methods. Always dispose created (disposable) objects.

```
// Add to create method
textureAtlas = new  TextureAtlas(Gdx.files.internal("spritesheet.atlas"));
animation = new Animation<>(0.1f, textureAtlas.getRegions());
TextureRegion textureRegion = textureAtlas.getRegions().get(0);
planesTextureWidth = textureRegion.getRegionWidth();
planesTextureHeight = textureRegion.getRegionHeight();

// Add to render method, at the end but inside of batch.begin() and batch.end()
elapsedTime += Gdx.graphics.getDeltaTime();
batch.draw(animation.getKeyFrame(elapsedTime, true),
           screenWidth / 4f - planesTextureWidth / 2f,
           screenHeight / 2f - planesTextureHeight / 2f);
```

Listing 12: Code for animation.

2. **Another animation with the same textures**. With a *TextureAtlas* object we can make several animations with the same images in memory. Add the following code to make another animation. Place this animation vertically in the middle and horizontally at 3/4 of the screen.

```
// Add to create method
TextureRegion[] rotateUpFrames = new TextureRegion[10];
for (int i = 0; i < rotateUpFrames.length; i++) {
   rotateUpFrames[i] = textureAtlas.findRegion(String.format("00%02d", i + 1));
}
rotateUpAnimation = new Animation<>(0.1f, rotateUpFrames);

// add to render method the necessary actions
```

Listing 13: Code for another animation.

3 Project P2Faces - user input, accelerometer, camera and Box2D

This is a multi-screen project that shows the use of user input, accelerometer, camera and Box2D library.

3.1 libGDX project and asset loader

1. **Create project and import resource files**. Create a new libGDX project - named *P2Faces* and leave the name of Game class as MyGdxGame. For the moment we will build a single screen app in the normal way. Latter we will adapt to enable multi-screens. Copy the following files to the *android/assets* directory of the project: *face_box_tiled.png*, *face_circle_tiled.png*, *face_triangle_tiled.png* and *face_hexagon_tiled.png*.

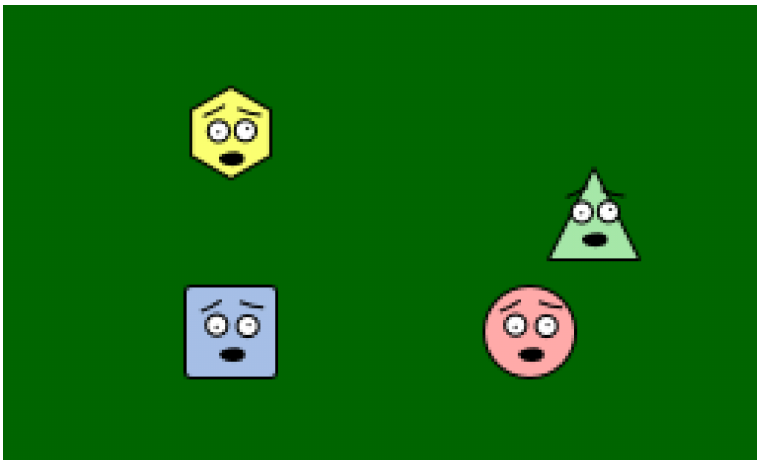


Figure 4: P2faces screen - Triangle moved by the accelerometer.

2. **Create an Assets Loader class**. In general, in a game, we load several *assets* into memory at the beginning. To help that, we will create an Asset Loader class. Create the *AssetsLoader* class that will load the 4 images, build *Texture* objects and create animations with this Textures. Use, and complete, the code in Listing 14.

```
public class AssetsLoader extends BaseAssetsLoader {
   // externally used textures and animations
   Animation<TextureRegion> faceBoxAnimation, faceCircleAnimation,
                            faceHexAnimation, faceTriAnimation;

   // load assets to textures and build animation
   AssetsLoader() {
      addDisposable(new Texture(Gdx.files.internal("face_box_tiled.png")));

      faceBoxAnimation = buildAnimationFromTexture((Texture) getDisposable(0),
                          32, false, false, Animation.PlayMode.LOOP_PINGPONG);
      ...
   }
}

class BaseAssetsLoader {
   private List<Disposable> disposableResources = new ArrayList<>();

   void addDisposable(Disposable disposable) {
      disposableResources.add(disposable);
   }

   Disposable getDisposable(int index) {
      return disposableResources.get(index);
   }

   // Extract square textureRegions from texture and build animation with them
   static Animation<TextureRegion> buildAnimationFromTexture(Texture texture,
             int textRegSize, boolean flipHorizontally, boolean flipVertically,
             Animation.PlayMode playMode) {
      int numberInWidth = texture.getWidth() / textRegSize;
      int numberInHeight = texture.getHeight() / textRegSize;
      int numberOfTextureRegions = numberInWidth * numberInHeight;

      TextureRegion[] texRegions = new TextureRegion[numberOfTextureRegions];
         for (int i = 0, x = 0, y = 0; i < texRegions.length; ++i) {
         texRegions[i] = new TextureRegion(texture,
                x * textRegSize, y * textRegSize, textRegSize, textRegSize);
         texRegions[i].flip(flipHorizontally, flipVertically);
         if (++x > numberInWidth) {
```

```
            x = 0;
            ++y;
         }
      }

      Animation<TextureRegion> animation = new Animation<>(0.5f, texRegions);
      animation.setPlayMode(playMode);
      return animation;
   }

   void dispose() {
      for (Disposable disposableResource : disposableResources) {
         disposableResource.dispose();
      }
   }
}
```

Listing 14: AssetsLoader code.

3. **Load the assets and play the animations**. Create an Asset Loader object in the *create* callback method and call the dispose action on *dispose* callback method. Add the four animations to the screen in the *render* callback method, in the same way as in previous project, but locate them in a way to slit the screen horizontally and vertically, having each image 300x300 pixels (use draw method with width and height).

3.2 User Input - Events

1. **Add user input events processing capacity**. To receive input from the user it is necessary to implement the *InputProcessor* interface. This interface is described in Listing 15. To do that, your game class must implement that interface (it is one way to do that). After that, choose "Implement methods", which is the action that the compiler suggests - this will add default implementations for interface methods. You must also register our *InputProcessor* (which in this case is your game class) on the *create* callback method - see code in the same Listing.

```
public interface InputProcessor {
   // Called when a key was pressed
   boolean keyDown(int keycode);

   // Called when a key was released
   boolean keyUp(int keycode);

   // Called when a key was typed
   boolean keyTyped(char character);

   // Called when the screen was touched, coords from the upper left corner
   boolean touchDown(int screenX, int screenY, int pointer, int button);

   // Called when a finger was lifted, coords from the upper left corner
   boolean touchUp(int screenX, int screenY, int pointer, int button);

   // Called when a finger was dragged, coords from the upper left corner
   boolean touchDragged(int screenX, int screenY, int pointer);

   // Called when the mouse was moved (for devices with mouse)
   boolean mouseMoved(int screenX, int screenY);

   // Called when the mouse wheel was scrolled (for devices with mouse)
```

```
    boolean scrolled(int amount);
}

// add to create callback method
Gdx.input.setInputProcessor(this);
```

Listing 15: Interface InputProcessor.

2. **Control *faceBoxAnimation* with user screen touch**. Redefine the *InputProcessor* interface callback methods, so that when the user is touching the screen, the *faceBoxAnimation* only presents the first image (use a flag to register the press down/up state, and get that image with *assetsLoader.faceBoxAnimation.getKeyFrames()[0]*). The animation should be rendered at the touch location, while pressing.

3.3 Accelerometer

1. **Control elements with the Accelerometer**. Here we want to use the accelerometer to control the triangle image/ animation. We want the behaviour like if the animation has weight - so when we put the device tilted for one side, the animation should move to the lowest part of the screen, in relation of the device 3D position.

   First, to use the accelerometer we must check on *create* callback method if that sensor is available in the host device. In Listing 16 shows the code to check for the availability of the accelerometer and also an auxiliary method to read the accelerometer values and update the animation coordinates. It is suggested that you log on debug messages the values read from the accelerometer to have a perception of them - focus on X and Y values. Set the DEBUG level for the logger. When all is OK, you can revert the debug level to INFO.

   Then you must update the animation coordinates, in the render method, with the values from the accelerometer. Do it with the auxiliary method, but you must adjust the update to animation coordinates inside the auxiliary method, to set the correct behaviour (take into account that in some functionalities, the coordinates are related to the main device orientation). Finally, add the final code to *render* callback method and adjust the coordinates of the animation to use the fields *triangleX* and *triangleY*.

   When the user presses down on the screen, set the animation coordinates back to its starting position.

```
// on create callback method
canUseAcel =  Gdx.input.isPeripheralAvailable(Peripheral.Accelerometer);

// auxiliary method
private void processAccelerometer() {
   float y = Gdx.input.getAccelerometerY();
   float x = Gdx.input.getAccelerometerX();

   // triangleX and triangleY are the coords of the triangle animation
   if (Math.abs(x) > 1)
      triangleX += x;

   if (Math.abs(y) > 1)
      triangleY += y;
}
```

```
// on render callback method
...
if(canUseAcel)
   processAccelerometer();

batch.begin();
...
```

Listing 16: Code to process accelerometer.

3.4 Setting an entry screen

1. **Setting an entry screen**. Now, he want to handle three screens, one new initial screen, our faces screen and a new Box2d screen. From the initial screen we want to enable to select the other two screens. Therefore, the entry screen should show two images, at its center and with 300x300 pixels each, and when the user touches one of them the *app* should activate the respective screen. The images should be the first image of faceBoxAnimation and faceHexAnimationon and they should activate the face screen and the Box2d screen, respectively. So, for now we will build: the main entry point class that is the *MyGdxGame* class; the initial screen class, which will be the *InitialScreen* class; the faces screen class, which will be a new *GameScreenFaces* class; and the Box2d screen class, which will be a new *GameScreenBox2d* that we will build in future sections. To keep the initial class, create a copy of *MyGdxGame* (the main class) to a class with the name *GameScreenFaces*. Hence, now, the *MyGdxGame* will be only an screen class loader. It should start by loading the the initial screen and this one will be responsible to load the *GameScreenFaces* and/or the *GameScreenBox2d*.

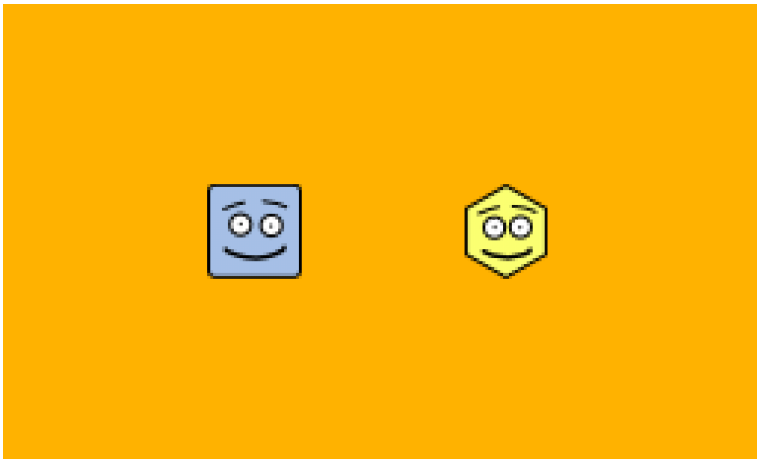


Figure 5: P2faces initial screen.

Replaces the code in *MyGdxGame* with the code in Listing 17.

```
// Game implements ApplicationListener (which implements ApplicationListener)
// This class setScreen for the several Screens
public class MyGdxGame extends Game {
   AssetsLoader assetsLoader;

   @Override
   public void create() {
      assetsLoader = new AssetsLoader();
```

```
        setInitialScreen();
    }

    void setScreenFaces() { setScreen(new GameScreenFaces(this)); }

    void setScreenBox2D() { setScreen(new GameScreenBox2D(this)); }

    void setInitialScreen() { setScreen(new InitialScreen(this)); }

    @Override
    public void dispose() { assetsLoader.dispose(); }

    static void log(String message) { Gdx.app.log("MyGdxGame", message); }
}
```

Listing 17: Multi-screen main class

Create the initial screen class with the name *InitialScreen*, in a way to show an orange (or other colour) background, with the first image of faceBoxAnimation and faceHexAnimation (with 300x300 pixels) at the center of the screen. The class should implement Screen and InputProcessor and in its constructor to receive the MyGdxGame class and save it locally. The class should display the two images and in the touchDown method should use the Rectangle class to check if the user touched (touchDown) the area of one of the images. Use the code in Listing 18 - to start you can comment the part related to *ScreenBox2D*. Add the unimplemented methods and complete the code in *render* method.

```
public class InitialScreen implements Screen, InputProcessor {
    private final float FIGURESIZE = 300f;
    ...

    InitialScreen(MyGdxGame mainClass) {
        batch = new SpriteBatch();
        this.mainClass = mainClass;
        this.assetsLoader = mainClass.assetsLoader;
        Gdx.input.setInputProcessor(this);
        faceBoxX = Gdx.graphics.getWidth() / 3f - FIGURESIZE / 2;
        faceHexX = 2 * Gdx.graphics.getWidth() / 3f - FIGURESIZE / 2;
        faceBoxY = Gdx.graphics.getHeight() / 2f - FIGURESIZE / 2;
        faceHexY = faceBoxY;
    }

    @Override
    public void render(float delta) {
        Gdx.gl.glClearColor(1, 0, 0, 1);  // set colour
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
        ... // show the images
    }

    @Override
    public boolean touchDown(int screenX, int screenY, int pointer, int button){
        MyGdxGame.log("touchDown...");
        Rectangle bounds= new Rectangle(faceBoxX,faceBoxY,FIGURESIZE,FIGURESIZE);
        if (bounds.contains(screenX, screenY)) {
            MyGdxGame.log("touchedDown on FaceBox image...");
            mainClass.setScreenFaces();
        } else {
            bounds = new Rectangle(faceHexX, faceHexY, FIGURESIZE, FIGURESIZE);
            if (bounds.contains(screenX, screenY)) {
                MyGdxGame.log("touchedDown on FaceHex image...");
                mainClass.setScreenBox2D();
            }
```

```
        }
        return false;
    }
    ...
}
```

Listing 18: Initial screen class

The *GameScreenFaces* class, now, should implement *Screen* (and *InputProcessor*) and no longer extend *ApplicationAdapter*. Therefore, you must add unimplemented methods of *Screen* interface. Turn the *create* method in the class constructor, receiving an *MyGdxGame* object and saving it into a class field. In the constructor remove the creation of the *AssetsLoader*. Turn the *render* method in the *render(float delta)* method. Check the code in Listing 19.

Now, the *app* will start with the *InitialScreen* and when the user touches the faceBox figure the *app* will change to *GameScreenFaces* screen.

```
public class GameScreenFaces implements Screen, InputProcessor {
    ...
    boolean canUseAcel =
                Gdx.input.isPeripheralAvailable(Input.Peripheral.Accelerometer);

    GameScreenFaces(MyGdxGame mainClass) {
        this.mainclass = mainClass;
        this.assetsLoader = mainClass.assetsLoader;
        batch = new SpriteBatch();
        Gdx.input.setInputProcessor(this);
        ...
    }

    @Override
    public void render(float delta) {
        Gdx.gl.glClearColor(0, 0.4f, 0, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

        if (canUseAcel) {
            processAccelerometer();
        }
        ...
    }
    ...
}
```

Listing 19: GameScreenFaces class.

2. **Return to initial screen with Android back button**. From the *GameScreenFaces* we want to return to initial screen, by pressing the Android back button. Catch the Android back button press events and call the set screen method from the main class to select the initial screen, like in Listing 20. Now, if you run the app and press that button, the app will be suspended. That occurs because the Android back button events are being processed as usual, by the Android code. To solve this, add an empty implementation of method *onBackPressed* on the *AndroidLauncher* class in the Android module.

```
@Override
public boolean keyDown(int keycode) {
    if(keycode == Input.Keys.BACK){
        log("back pressed");
        mainclass.setInitialScreen();
```

```
            return true;
        }
        return false;
    }
```

Listing 20: Go back with Android back button press event.

4 Box2d world - Inside P2Faces project

It is just one more screen in the P2Faces project and not a new project. It is separated, in the text, because it approaches the **Box2d** library, which is a new theme.

In Figure 6 it shown a final image for this world.

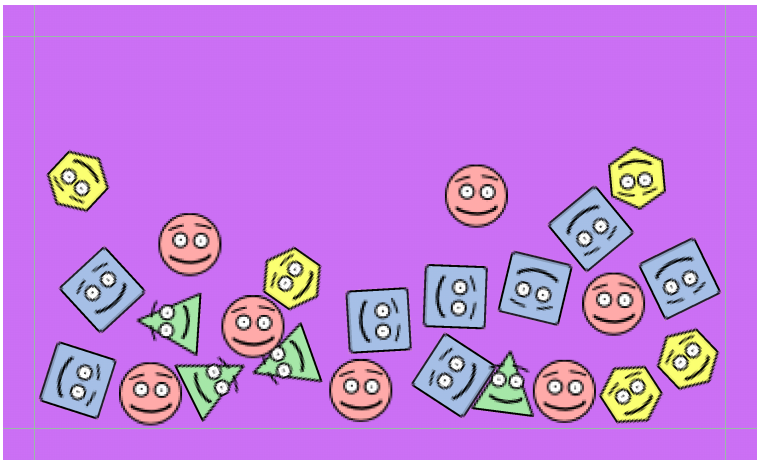


Figure 6: Box2D - World of objects/forms.

4.1 Build Box2d world

1. **Create *GameScreenBox2D* class and prepare it**. Create a new *GameScreen-Box2D* class to support a Box2D world. This class should implement *Screen* and *InputProcessor* interfaces. Add unimplemented methods. Create a constructor that receives a MyGdxGame object and saves it in a class field (it will be used to return to initial screen and get assets). The constructor should also prepare a Spritebatch and set InputProcessor. Set the background colour as violet, in the render method. Set Android back button action to return to the initial screen. In the *InitialScreen* class activate/uncomment the code to activate *GameScreenBox2D* screen and test it.

2. **Create the Box2D world**. Box2D is a library of 2D physics (physics engine) one the most popular in 2D games and used in most programming languages. To use the Box2D engine we may create a World object (com.badlogic.gdx.physics.box2d.World). Creating such object will internally call the Box2D.init(), which is the real start up action or the Box2D engine. We will also create a *Box2DDebugRenderer* object as it is useful to test the application in *debug* mode. Add the code in Listing 21 to create these two objects in the class constructor.

```
// add to constructor
// Builds a Box2D world
// First argument (Vector) sets the horizontal and vertical gravity forces
// Second argument tells to render all bodues, even inactive bodies
world = new World(new Vector2(0, 0), false);

// Builds a Box2DDebugRenderer object
debugRenderer = new Box2DDebugRenderer();
```

Listing 21: Create World code.

3. **Add a Camera**. To render the world in an easy way we need to use a *Camera* (we will not use the camera functionalities in this project). Add the code in Listing 22 to the class. For now, the result should still be a beautiful violet empty world.

```
// on class constructor
// build OrthographicCamera
camera = buildCamera();
// Update the batch with our Camera's view and projection matrices
batch.setProjectionMatrix(camera.combined);

// class method
private OrthographicCamera buildCamera() {
   OrthographicCamera camera = new OrthographicCamera(
                            Gdx.graphics.getWidth(), Gdx.graphics.getHeight());

   // Set camera's position in the center of the screen
   camera.position.set(camera.viewportWidth * .5f,
                        camera.viewportHeight * .5f, 0f);

   // Update and return the camera
   camera.update();
   return camera;
}
```

Listing 22: Create Camera object.

4. **Render the world**. To render the world in *Debug* mode add the code below. This code defines how the world is updated (e.g., 50 or 60 frames/second).

```
// Class Atributes
private static final float WORLD_STEP = 1 / 60f;
private static final int WORLD_VELOCITY_ITERATIONS = 6;
private static final int WORLD_POSITION_ITERATIONS = 2;

// Add to render callback method
// show debug information
debugRenderer.render(world, camera.combined);
//  here batch begin/end
// Make a world step
world.step(WORLD_STEP, WORLD_VELOCITY_ITERATIONS, WORLD_POSITION_ITERATIONS);
```

Listing 23: World render and step.

5. **Delimit the world space**. The world should be the screen rectangle, delimited by 4 static areas/objects/walls. Static bodies are perfect for ground, walls, and any object which does not need to move. Copy the code below to the class and complete it. To better understand the code you should consult the Box2D manual. With the code (and your code), you should see something like in Figure 7 (without objects, for now).

```
// call createWorldBoundaries() on constructor

// class methods

// Create 4 boundary objects with thickness of .5f, limiting the device screen
// Build one static body at 100 of the top, bottom, left and right
private void createWorldBoundaries() {
  // bottom - .5f is half pixel
  createStaticBody(0, 100, camera.viewportWidth, .5f);
  // top
  createStaticBody(0, camera.viewportHeight - 100, camera.viewportWidth, .5f);
  // left
  createStaticBody(100, 0, .5f, camera.viewportHeight);
  // right
  ... // define this, to make thing easy to observe, set this to width / 4
}

private void createStaticBody(float x, float y, float width, float height) {
  BodyDef WallBodyDef = new BodyDef();
  WallBodyDef.position.set(new Vector2(x, y));
  Body wallBody = world.createBody(WallBodyDef);
  PolygonShape wallBox = new PolygonShape();
  wallBox.setAsBox(width, height);
  wallBody.createFixture(wallBox, 0.0f);
}
```

Listing 24: Create world boundaries.

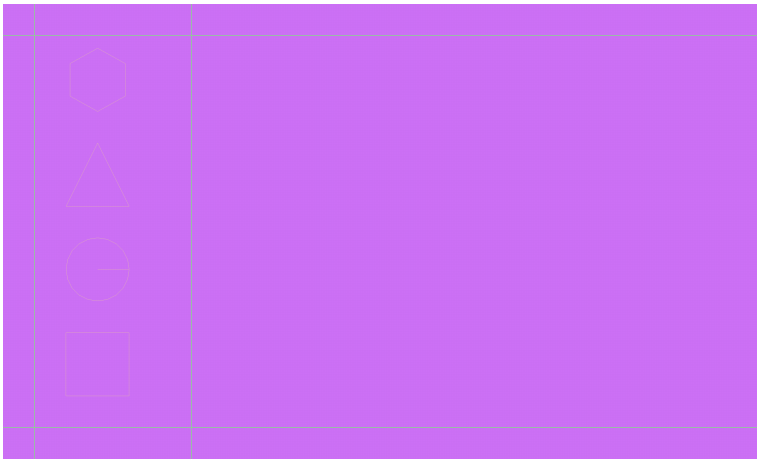


Figure 7: Box2D - World with delimiters and initial objects.

4.2 Create world objects

1. **Create object builders**. Here we will create **object builders** to allow us to create objects, either manually or dynamically. Copy the code below to the class. It defines an enumerate with the existing objects types and defines the methods to create body type objects of shapes: box, circle and hexagon. Add a method to create body type objects with a triangle shape.

```
// class constant with the figures size
private static final float FIGURE_SIZE = 200f;

// class member
public enum FormType {
   Box, Circle, Triangle, Hexagon
```

```
}

// class methods
private void createBoxBody(int screenX, int screenY) {
   PolygonShape box = new PolygonShape();
   box.setAsBox(FIGURE_SIZE / 2, FIGURE_SIZE / 2); // hx/hy  half-width/height
   createBody(screenX, screenY, box, FormType.Box);
}

private void createCircleBody(int screenX, int screenY) {
   CircleShape circle = new CircleShape();
   circle.setRadius(FIGURE_SIZE / 2);
   createBody(screenX, screenY, circle, FormType.Circle);
}

private void createHexBody(int screenX, int screenY) {
   PolygonShape hex = new PolygonShape();
   Vector2[] vertices = new Vector2[6];
   vertices[0] = new Vector2(0, FIGURE_SIZE / 2);       // top
   vertices[1] = new Vector2(0, -FIGURE_SIZE / 2);      // bottom
   vertices[2] = new Vector2(-FIGURE_SIZE * 0.4375f, FIGURE_SIZE * 0.2578125f);
   vertices[3] = new Vector2(-FIGURE_SIZE * 0.4375f, -FIGURE_SIZE * 0.2578125f);
   vertices[4] = new Vector2(FIGURE_SIZE * 0.4375f, FIGURE_SIZE * 0.2578125f);
   vertices[5] = new Vector2(FIGURE_SIZE * 0.4375f, -FIGURE_SIZE * 0.2578125f);
   hex.set(vertices);
   createBody(screenX, screenY, hex, FormType.Hexagon);
}

private void createBody(int screenX, int screenY,
                        Shape shape, FormType formType) {
   BodyDef bodyDef = new BodyDef();
   bodyDef.type = BodyDef.BodyType.DynamicBody;
   bodyDef.position.set(screenX, screenY);
   Body body = world.createBody(bodyDef);
   body.createFixture(createFixtureDef(shape));
   body.setUserData(formType);
}

private FixtureDef createFixtureDef(Shape shape) {
   FixtureDef fixtureDef = new FixtureDef();
   fixtureDef.shape = shape;
   fixtureDef.density = 1.0f;
   fixtureDef.friction = 0.0f;
   fixtureDef.restitution = 1;
   return fixtureDef;
}
```

Listing 25: Define world body shapes.

2. **Manually add some objects**. Here, we want to manually create initial objects. Use the code below to create one object of each our types. You should see the objects that are in Figure 7.

```
// add to constructor - to create initial objects
createBoxBody(300, 300);
createCircleBody(300, 600);
createTriangleBody(300, 900);
createHexBody(300, 1200);
```

Listing 26: Create initial bodies/shapes.

3. **Set world gravity and see objects falling and colliding**. Set the World gravity to Vector2(0, -20) and run the program. You should see the objects falling down

to the bottom of the screen and behave like physical objects. See a screen-shot in Figure 8.

```
// Add to in constructor
world.setGravity(new Vector2(0, -20));
```

Listing 27: Set gravity.

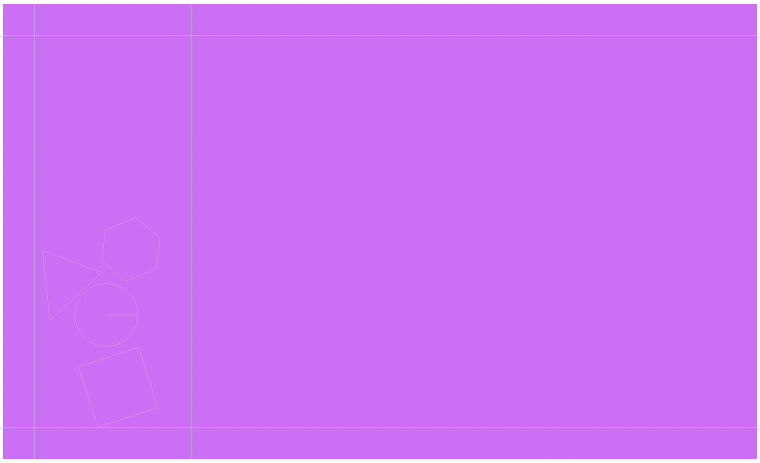


Figure 8: Box2D - World with gravity.

4. **Dynamically add objects to the world**. Use the *touchDown* callback method to create random objects at the touch position. Use the code below and test it. In Figure 9 you can see a screen-shot with some objects dynamically created.

```
// class field
RandomXS128 random = new RandomXS128(System.currentTimeMillis());

// class methods
public boolean touchDown(int screenX, int screenY, int pointer, int button) {
   log("touchDown on x:" + screenX + ", y " + screenY);
   int y = Gdx.graphics.getHeight() - screenY;
   log("insert object at x:" + screenX + ", y " + y);
   addRandomBody(screenX, y);
   return false;
}

private void addRandomBody(int screenX, int screenY) {
   switch (FormType.values()[random.nextInt(FormType.values().length)]) {
      case Box:
         createBoxBody(screenX, screenY);
         break;
      case Circle:
         createCircleBody(screenX, screenY);
         break;
      case Triangle:
         createTriangleBody(screenX, screenY);
         break;
      case Hexagon:
         createHexBody(screenX, screenY);
   }
}
```

Listing 28: Dynamically add objects.

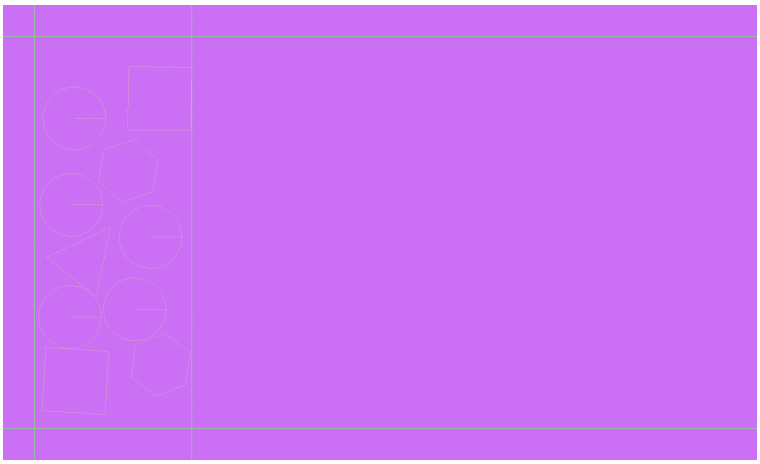


Figure 9: Box2D - World with some dynamically created objects.

4.3 Add animations to objects

1. **Add images/animation to the objects**. Here, we will add the respective images/animation to the objects. In the *render* callback method we must go through all the bodies of the world and render them with their images/animations. Use the code below. In Figure 10 you can see a screen-shot with the initial objects, after some time.

   You can see that the images do not follow th objects rotation. We will solve this in next point.

```
// class attribute, to support World objects - a permanent array
Array<Body> bodies = new Array<>();

// render method - render world objects with their animations
public void render(float delta) {
   Gdx.gl.glClearColor(0, 0, 1, 1);
   Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
   elapsedTime += delta;

   debugRenderer.render(world, camera.combined); // objects debug shape
   batch.begin();
   world.getBodies(bodies);
   for (Body body : bodies) {
      Object data = body.getUserData();
      if (data != null) {
         Animation<TextureRegion> animation;
         if (data == FormType.Circle) {
            animation = assetsLoader.faceCircleAnimation;
         }
         else if (data == FormType.Triangle) {
            animation = assetsLoader.faceTriAnimation;
         }
         else if (data == FormType.Hexagon) {
            animation = assetsLoader.faceHexAnimation;
         }
         else {
            animation = assetsLoader.faceBoxAnimation;
         }

         TextureRegion region = animation.getKeyFrame(elapsedTime, true);
         batch.draw(region, body.getPosition().x - FIGURE_SIZE / 2,
                            body.getPosition().y - FIGURE_SIZE / 2,
                            FIGURE_SIZE, FIGURE_SIZE);
      }
```

```
    }
    batch.end();
    world.step(WORLD_STEP, WORLD_VELOCITY_ITERATIONS, WORLD_POSITION_ITERATIONS);
}
```

Listing 29: Render world objects.

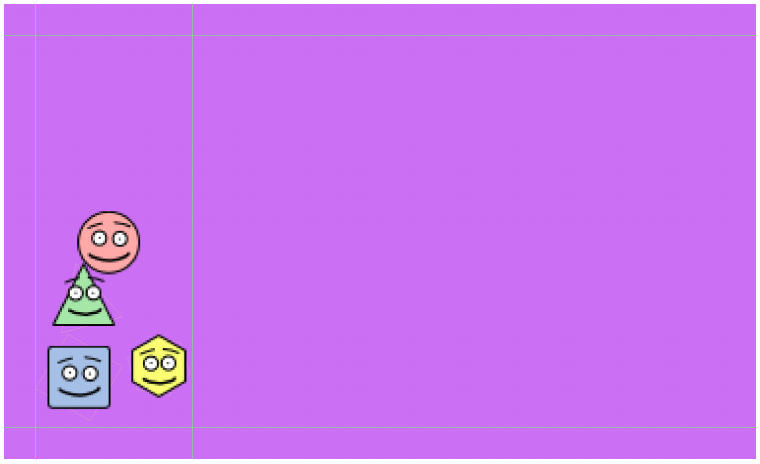


Figure 10: Box2D - World with initial objects, animations and gravity, after some time. Figures misaligned with their bodies.

2. **Align animation with object bodies**. To do that we must set object angle using the following method to draw the animation in the render method (replacing the existing draw call). Read carefully its documentation and only them use it. The final result should be a complete align between object bodies and their animations, as can be seen in Figure 11 and next ones.

```
// In render claback method call this method (instead of existing one)
public void draw (TextureRegion region, float x, float y,
                  float originX, float originY, float width, float height,
                  float scaleX, float scaleY, float rotation) {
```

Listing 30: Draw function with angle rotation.

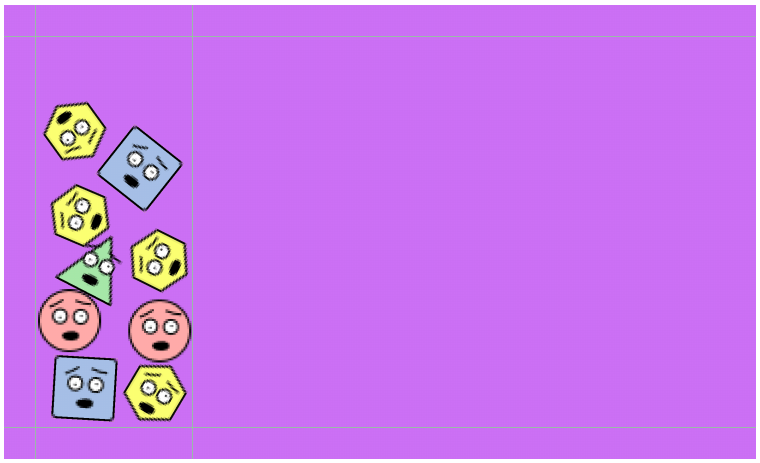


Figure 11: Box2D - World with objects with animations correctly shown.

4.4 Build an object fountain

1. **Build an object fountain**. Set a way to create 20 random objects at the middle and near the top, at the rhythm of 200ms (or other value, defined in a constant). Set the right delimiter object of the word to *width - 100*. You can see the result in Figures 12 and 13 and also in the following interesting video about bodies and shapes:
   `https://www.youtube.com/watch?v=oBcv_P_YDrw`.

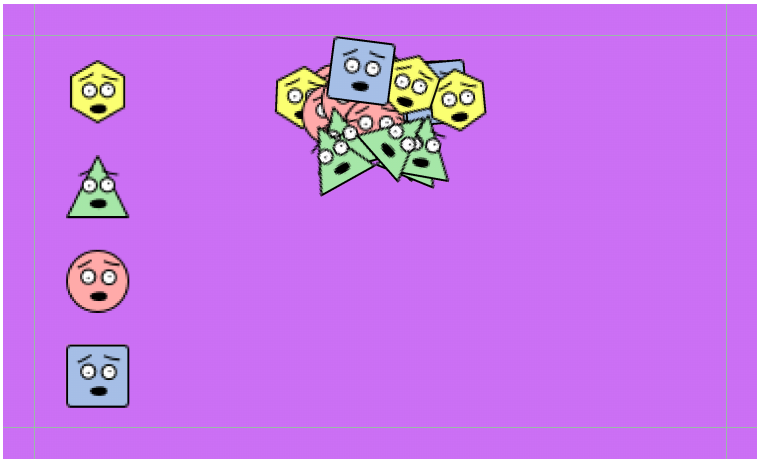
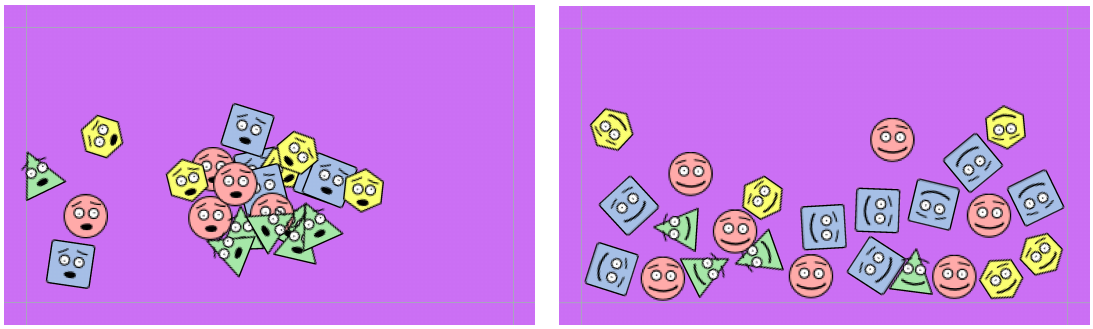


Figure 12: Box2D - World with an object fountain.



(a) After some time. (b) After more time.

Figure 13: World with an object fountain.

5 Project P3Cowboy - background parallax

This project shows the use of animations with Sprites, and the use of effect of parallax with multiple background images. In Figure 16 shows an image of this world.

5.1 libGDX project and asset loader

1. **Create project and set their files**. Create a new **libGDX** project, named **P3Cowboy**, in a directory with that name and using **Box2d** library.

   Copy the following files to the *assets* directory of the *android* module of the project: *cowboy_run_sprite.png*, *parallax_background_layer_back.png*, *parallax_background_layer_front.png* and *parallax_background_layer_mid.png*.

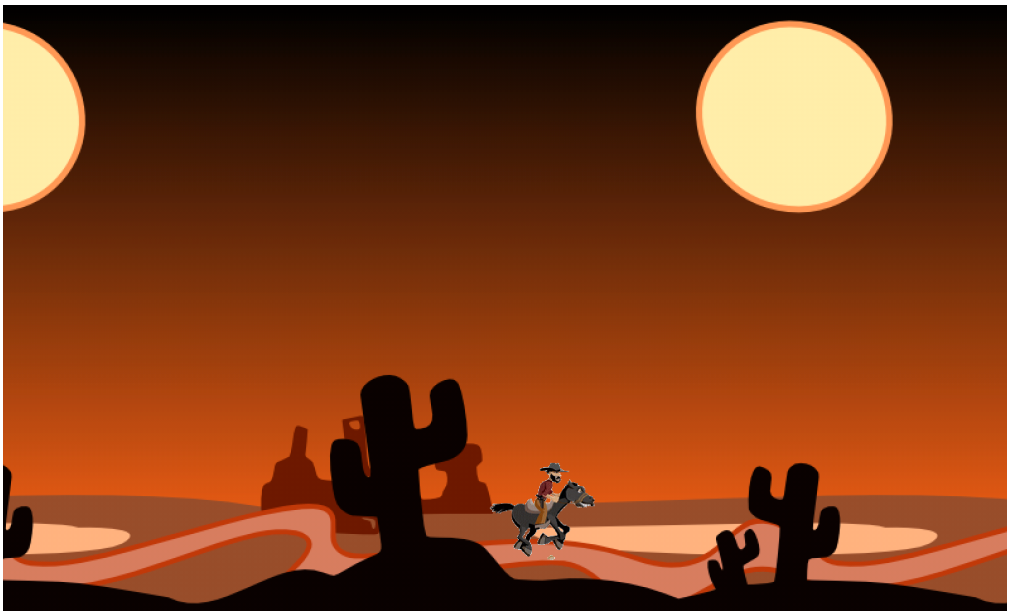


Figure 14: Cowboy world.

2. **Create the AssetsLoader**. Create an *AssetsLoader* class to build textures from the 4 files and to build the cowboy animation. Use, and complete, the code in Listing 31 - get a copy of *BaseAssetsLoader* class from previous project.

```
public class AssetsLoader extends BaseAssetsLoader {
   // externally used textures and animations
   Texture backTexture, midTexture, frontTexture;
   Animation<TextureRegion> cowboyAnimation;

   AssetsLoader() {
      Texture cowboyTexture;
      addDisposable(backTexture =
        new Texture(Gdx.files.internal("parallax_background_layer_back.png")));
      ...

      cowboyAnimation = buildAnimationFromTexture(cowboyTexture, 175,
                        true, false, Animation.PlayMode.NORMAL);
   }
}
```

Listing 31: AssetsLoader class.

5.2 Build base Cowboy scenario

1. **Make it a multi-screen project**. We will only use one screen, but we will prepare the project to be a multi-screen one. We request this action for two purposes: for you to practice and to prepare the project to be extended with any extra code that you may want to do.

   Prepare the game launcher class. Copy all the *MyGdxGame* class from previous project and overlap that class in current project with that code. Remove *setScreen* methods, except *setInitialScreen*, which should work with our single screen *GameScreenCowboy* class, by replacing the existing call with *GameScreenCowboy* as class constructor name and asking to create that class in the same package.

   Prepare the GameScreenCowboy class. As that class should implement the

Screen interface, add all unimplemented methods. This class should render the world objects, hence it typically contains the *world* and *camera* objects, but as its goal is so simple that it will neither have a World nor a Camera. For now, in the constructor: save the main class object, save a reference to the *assetsLoader* and create a *SpriteBatch*.

Create (or not) the Actors classes. Usually, in the basic structure of a game, it is necessary to create a hierarchy of classes to support the several kind of actors in the game. To do that we should define a base Actor class and define derived classes for existing specific kind of actors. In this tutorial, as the actors are very simple, we will not create the Actors class hierarchy.

2. **Set static background and cowboy images**. To start, we will set static background and cowboy images. In *render* callback method set as full screen background the *background_layer_back* image and place the first cowboy image horizontally in the middle and about 90 pixels vertically from be bottom. Run the program. You should see a world like in Figure 15.

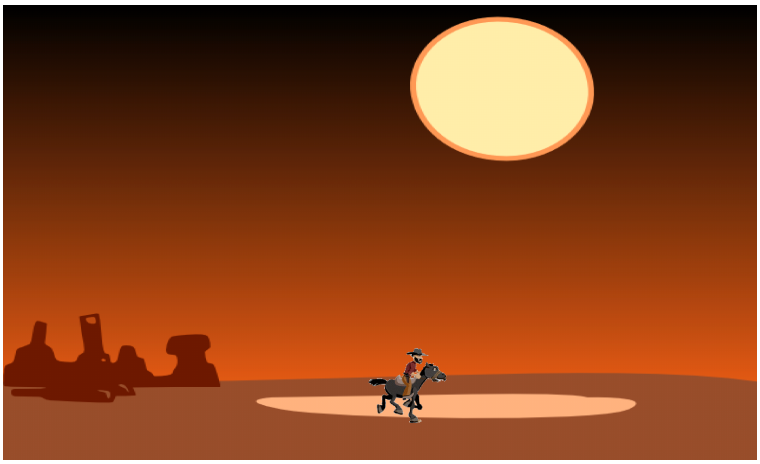


Figure 15: Initial cowboy world.

5.3 Build a parallax scrolling background

1. **Build a parallax scrolling background**. The effect*parallax scrolling background* is a web design technique, in which the background moves at a slower pace than that of the foreground. This results in a 3D effect as visitors scroll the scenario, adding a sense of depth and creating a more immersive experience. We will use a scenario with three layers and the cowboy, which should be viewed in this order: back, mid, cowboy, front. We will use for the scenario layers the respective parallax images.

   Build a scrollable class. All three background images should scroll to the left and at different paces. To support that movements we need a class that for each layer it keeps: the current X position, the pace and the image/texture. You should build the "Scrollable" class for that purpose, based on the code in Listing 32. The code is tailored for this example and we challenge students, who need this movement in their games, to build a more generic class (ex: supporting also vertical movements) or even a class hierarchy to support different behaviours.

```
public class Scrollable {
   private Texture texture;
   private Vector2 position, velocity;
   private int width, height;

   public Scrollable(Texture texture, float x, float y, int width, int height,
                                                          float scrollSpeed) {
      this.texture = texture;
      this.position = new Vector2(0, 0);
      this.velocity = new Vector2(scrollSpeed, 0);
      this.width = width;
      this.height = height;
   }

   public void update(float delta) {
      position.add(velocity.cpy().scl(delta));

      if (position.x + width < 0) {
         position.x += width;
      }
   }

   public void draw(SpriteBatch batch) {
      batch.draw(texture, position.x, 0, width, height);
      batch.draw(texture, position.x + width, 0, width, height);
   }
}
```

Listing 32: Scrollable class for parallax.

Create layer scrollable objects and render them. For each layer create a Scrollable object with the layer definitions: initial (x,y), width and height, and the velocity. Name them: scenarioBack, scenarioMiddle and scenarioFront. The velocity should be -5, -20 and -75, for the back, middle and front layers, respectively. Use the code in Listing 33, and complete it for the remaining layers. Run the *app*, you should see the scenario layers moving to the left at different paces.

```
// class constructor, for each layer create a Scrollable object with their data
scenarioBack = new Scrollable(assetsLoader.backTexture, 0, 0,
                      Gdx.graphics.getWidth(), Gdx.graphics.getHeight(), -5f);

// on render method, repeat for other layers
scenarioBack.update(delta);

batch.begin();
scenarioBack.draw(batch);
batch.end();
```

Listing 33: Parallax layers

2. **Add cowboy animation**. Replace the static cowboy image, with the cowboy animation, which should be viewed between the middle and front layers. The result should be similar to the Figure 16.

3. **Correct background layers visualization**. As you can check, the layers are shown with distortion. That happens because the images stretch, to the screen dimensions, is not proportional in relation to width and height. Here we want that you to correct this, showing these layers in a proportional way, which means with not distortion. The final result should be similar to the Figure 14.

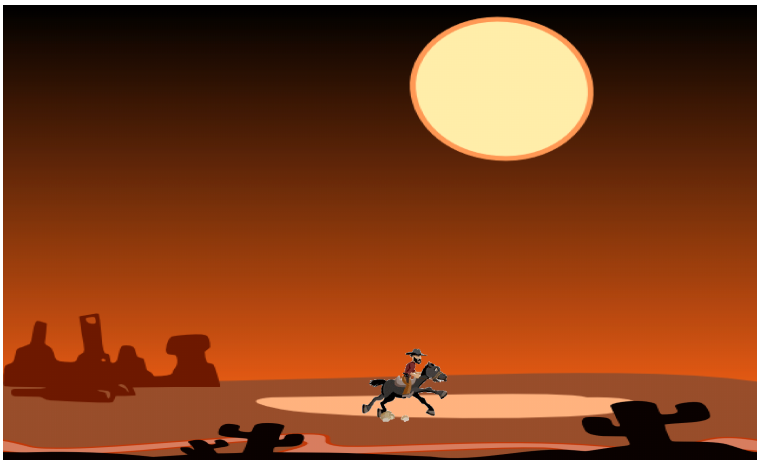


Figure 16: Cowboy world - distorted.

6 Project P4LordOfTanks - audio

This project shows the use of audio. Figure 17 contains its screen-shot.

Figure 17: *Lord Of Tanks* world.

6.1 libGDX project and Assets Loader

1. Create a new libGDX project, named **P4Tanks**, in a directory with that name and **Box2D** library. Copy the following files to the *assets* directory of the *android* module of the project:

   - images - *back.png*, *notes.png*, *tank.png*;
   - text fonts - *text.fnt*/*text.png*, *shadow.fnt*/*shadow.png*;
   - audio - *explosion.ogg*, *wagner_the_ride_of_the_valkyries.ogg*.

2. **Create the *AssetsLoader***. Create the assets loader to load the images, the sounds and the fonts. Use the code in Listing 34. Set the font scale as you wish.

```
class AssetsLoader extends BaseAssetsLoader {
  // externally used textures and animations
  final Texture backTexture, notesTexture, tankTexture;
  final Sound explosion, music;
  final BitmapFont font, shadow;

  AssetsLoader() {
    // Textures
    addDisposable(backTexture= new Texture(Gdx.files.internal("back.jpg")));
    addDisposable(notesTexture= new Texture(Gdx.files.internal("notes.png")));
    addDisposable(tankTexture= new Texture(Gdx.files.internal("tank.png")));
    addDisposable(textTexture= new Texture(Gdx.files.internal("text.png")));
    addDisposable(shadowTexture= new Texture(Gdx.files.internal("shadow.png")));

    // Sounds, LibGDX supports audio formats: ogg, mp3 and wav
    String soundName = "explosion.ogg";
    addDisposable(explosion= Gdx.audio.newSound(Gdx.files.internal(soundName)));
    soundName = "wagner_the_ride_of_the_valkyries.ogg";
    addDisposable(music = Gdx.audio.newSound(Gdx.files.internal(soundName)));

    // Fonts
    addDisposable(font = new BitmapFont(Gdx.files.internal("text.fnt")));
    font.getData().setScale(...);
    addDisposable(shadow = new BitmapFont(Gdx.files.internal("shadow.fnt")));
    shadow.getData().setScale(...);
  }
}
```

Listing 34: Assets Loader.

3. **Build a multi-screen project.** Build a multi-screen project, with an initial screen class with the name *GameScreenLordOfTanks*. That class should also implement *InputProcessor*, as we we will use two buttons (areas) to play two sounds. Add all unimplemented methods. Proceed as in previous project. In the constructor: save *mainClass* object, *assetsLoader*, build *SpriteBatch* and also set input processor with current object, create a *ShapeRenderer* object (*shapeRenderer = new ShapeRenderer()*) and create the buttons to play sounds, calling the *defineButtons* method presented in Listing 35 (use a factor of 4), for the images *tank.png* and *notes.png*.

   In fact, we create two rectangles around the images, then we'll set the *touchUp* event to receive the user's touch and start playing the sounds.

```
private void defineButtons() {
   // 2 rectangles to used around the textures in order to built the buttons
   float tankWidth = assetsLoader.tankTexture.getWidth() * factor;
   float tankHeight = assetsLoader.tankTexture.getHeight() * factor;
   tankButton = new Rectangle(
                        Gdx.graphics.getWidth() / 4f - tankWidth / 2f,
                        3 * Gdx.graphics.getHeight() / 5f - tankHeight / 2f,
                        tankWidth, tankHeight);

   float notesWidth = assetsLoader.notesTexture.getWidth() * factor;
   float notesHeight = assetsLoader.notesTexture.getHeight() * factor;
   notesButton = new Rectangle(
                        3 * Gdx.graphics.getWidth() / 4f - notesWidth / 2f,
                        3 * Gdx.graphics.getHeight() / 5f - notesHeight / 2f,
                        notesWidth, notesHeight);
}
```

Listing 35: Create sound buttons.

4. **Draw background, buttons images, button areas and text**. In the *render* callback method, render the images, the rectangles and display the text, with the font and the shadow, on the screen. Make the changes to fill the *shapeRenderer*, of the *tank* and the *notes* images, with solid colour and add to them a nice border of 50 pixels, like in Figure 17.

```
public void render(float delta) {
  batch.begin();

  // draw background
  batch.draw(assetsLoader.backTexture, 0, 0,
             Gdx.graphics.getWidth(), Gdx.graphics.getHeight());

  // Draw shadow and text
  assetsLoader.shadow.draw(batch, "Touch for Sound!",
        Gdx.graphics.getWidth() / 2f - 150 + 5,
        200 + assetsLoader.font.getCapHeight() + 5, 400, Align.center, false);
  assetsLoader.font.draw(batch, "Touch for Sound!",
       Gdx.graphics.getWidth() / 2f - 150,
       200 + assetsLoader.font.getCapHeight(), 400, Align.center, false);

  // draw tank and notes image
  batch.draw(assetsLoader.tankTexture, tankButton.x, tankButton.y,
             tankButton.getWidth(), tankButton.getHeight());
  batch.draw(assetsLoader.notesTexture, notesButton.x, notesButton.y,
             notesButton.getWidth(), notesButton.getHeight());

  batch.end();

  // draw delimiter lines
  shapeRenderer.begin(ShapeRenderer.ShapeType.Line);
  shapeRenderer.setColor(tankColor);
  shapeRenderer.rect(tankButton.x, tankButton.y,
                     tankButton.width, tankButton.height);
  shapeRenderer.setColor(notesColor);
  shapeRenderer.rect(notesButton.x, notesButton.y,
                     notesButton.width, notesButton.height);
  shapeRenderer.end();
}
```

Listing 36: Render code.

6.2 Play sound

1. To finish, copy the code below to play the sounds and to stop them. Run the program and try the sounds, touching the *tank* and *notes* areas.

```
public boolean touchDown(int screenX, int screenY, int pointer, int button) {
   // Mouse Y is inverted
   screenY = Gdx.graphics.getHeight() - screenY;

   log("TouchUp, x:" + screenX + " y:" + screenY);  // from previous projects

   if (notesButton.contains(screenX, screenY)) {
      log("Touch on Notes button");
      assetsLoader.music.stop();
      assetsLoader.music.play();
   } else if (tankButton.contains(screenX, screenY)) {
      log("Touch on Tank button");
      assetsLoader.explosion.play();
   } else {
      assetsLoader.music.stop();
      assetsLoader.explosion.stop();
```

```
    }

    return false;
}
```

Listing 37: On touch, play/stop sound.