Football Game

# Summary

The game is a simple football game with two players kicking a ball and trying to score in the time limit. The application can run on PC and Android phones.



# Technology used

For the development I’ve chosen LibGDX as required. This game also utilizes physics, so Box2D was also added into the dependencies, as well as gdx-freetype, which is a library that’ll allow us to generate different sized fonts from a true-type font at runtime.

# Sub projects inside the main project

* /android – the necessary skeleton for the Android version of the project
* /desktop – the necessary skeleton for the PC version of the project
* /core – this is the common part, where the actual game logic is implemented

# Implementation details

## JeuDeFootball

This is the main entrypoint of the Core. It’s responsible for changing the current [Screen](#_Screen), updating the logic of them, and renders them. It also overrides the dispose() function to release the resources from the memory after closing the game. When changing the [Screen](#_Screen), the class will set a [TransitionListener](#_TransitionListener) to them, which is a utility interface, that’ll help in changing the screen.

## Utility classes

We have three utility classes for different assets of the game. These are the following:

### TextureUtil

This is a Singleton class, which holds all the textures and fonts of the game. The main purpose of this class is to give all other classes a common place to access the textures, as well as the possibility to release all of them from the memory. The textures are stored in a HashMap, with a String key, so it’s easier to access them. The project only uses one Font, so there isn’t a Map for fonts. Nevertheless, the Font itself is generated there as well, in the load() function.

### SoundUtil

This is also a Singleton class, which will hold all the sounds. As the loading of the sounds could take some time, we’ll instantiate the class on the splash screen, hence we won’t notice any lag while in a match. In contrast of TextureUtil, which will only provide the textures for the calling classes, SoundUtil will also play the sounds for them.

### ModelUtil

ModelUtil is a simple class with a static function, which converts a file, containing the coordinates of a bounding shape to a list of 2d vectors, and scales them according to the window’s aspect ratio

## Screens

Screens are classes responsible for rendering different stages of the game. Their functions are externalized via the Screen interface. [JeuDeFootball](#_JeuDeFootball) is responsible for changing, updating and rendering the screen.

### Screen

Screen is a common interface between different screens. It defines a tick(), and a draw() function, and accessors for a [TransitionListener](#_TransitionListener).

### Splash

This is the first screen the user encounters. It shows a single image for 3 seconds, then it’ll transition to the [Menu](#_Menu) screen.

### Menu

This is the main menu, with 3 different options for the user to interact with. We define 3 Buttons for these options. The first button takes the player to the [Game](#_Game) in [Single Player Mode](#_Game_Logic), the second will start the [Two Player Mode](#_Game_Logic), and the last button will quit from the game.

### Game

This is the game screen, the behaviour is explained in the Game Logic section.

### Button

This is an abstract class representing a button. In the constructor a position, a width and a height is given. We store the extremes in Vector2 as well. It has a *boolean isInside(float, float)* function which returns whether the given coordinate is inside the bounds of the button. void *onClick()* is an abstract function, which the implementing classes should implement, this is what’ll be called if the button is clicked.

## Entities

Entities are the dynamic entities of the game. They are updated in every frame, according to their behaviour.

### Entity

This is the abstract class for all the dynamic entities in the game. Since most of the entity related fields are common, this will implement most of the behaviour. It has the following fields:

* *Vector2 startPos* – The starting point of an entity, it won’t be changed after initialization.
* *Vector2 position* – The current position of the entity on the screen
* *float width, height* – The size of the entity. In the current version of the game, all entities are circular, but for future expandability there is both width and height.
* *Texture texture* – The texture of the entity. It’s loaded in the constructor from the [TextureUtil](#_TextureUtil) using a String parameter
* *float maxSpeed* – The entities speeds are limited to be realistic
* *float restitution* – The restitution of the physical body of the entity
* *Controller controller* – Different [Controllers](#_Controllers) could be allocated for an entity
* *Body body* – The Box2D body of the entity, which will be used in physics simulations
* *float restitution* – The restitution of the Body

The functions (except getters and setters) are the following:

* *reset()* – Resets the entity to the starting position, also resets the linear velocity to zero
* *draw(SpriteBatch)* – Draws the entity with the given SpriteBatch. The locations are the middle of the Entity, so the drawing coordinates are translated with half of the size.
* *tick()* – This function will update the controller if exists, and updates the position according to the physical location.
* *createBody(World)* – Given a Box2D World parameter, this function will initialize the Body of the Entity. The bodies are CircleShape fixtures with a density of 1, and restitution according to the class field. The bodies are also fixed in rotation to prevent strange behaviour.
* *accelerate(float, float)* – Given an impulse direction, this function will accelerate the physical Body of the object, and limits the velocity to the maxSpeed.

### Ball

The Ball is a child of Entity, and it’s responsible for the ball in the game. It’s size is 1/85th of the window’s width, the restitution is changed to 0.5f, and after the Body creation, it’s only fixtures userData is set to [**BALL**](#_EntityTypes).

### LeftPlayer and RightPlayer

These are the classes for the players, the only difference is their starting location and texture. Their size is the 1/40th of the playing field. The userData of the fixture is set to [**PLAYER**](#_EntityTypes)

### EntityTypes

This is an enum containing the different types of userData a fixture could take. The values are the following:

* **BALL** – For the ball
* **PLAYER** – For the players
* **LEFT\_GOAL** – For the left goal
* **RIGHT\_GOAL** – For the right goal

## Controllers

These are the different controllers to control the Entities.

### Controller

The common utility interface for the Controllers. It’s method is the *update(*[*Entity*](#_Entity)*)*  which will update the [Entity](#_Entity) in the parameter according to the Controller’s logic

### PlayerController

The Controller for user control. As a constructor parameter, it’ll get four integers according to the keycodes to move up, down, left and right, a Vector2 defining the centre of the touch controlling surface, and a radius, in which that controller will respond to touchdown events. The radius’ square is calculated in the constructor to avoid using square roots in the distance check. A speed is also present which designates the acceleration speed of the player. The Controller’s logic is the following:

If the up button is pressed, move the player up. If not, and the down key is pressed, move the player down. This is to make sure that we don’t give opposing acceleration vectors to the Entity in the same frame. Then the left and right buttons are checked as well in the same manner.

After keyboard checks, we’ll check the touched points. If the touch start is inside the controlling radius, we check the current position of the touch event, and the player will accelerate according to the Vector2 from the the controllerCenter to the touch event, normalized and scaled to speed.

### CPUController

This is a simple AI for a CPU player. Its logic is very simple. In the constructor, it’ll get an Entity referencing the Ball, and a Vector2 target. The AI will try to target the location of the ball opposite to the target.

Targett

Ball

Spot to target

Direction of movement

## Listeners

### TouchListener

This is a Singleton Listener extending LibGDX InputAdapter. The sole purpose of this class is to store touch events. It’ll store up to 5 different touch/mouse inputs, with the current location, and starting location. It also has a TouchInfo inner static class to give a structure to these. As it’s a Singleton, it’s easy to get the instance and hence the touch events in any classes.

### TransitionListener

This is a utility interface which purpose is to provide an interface for [Screen](#_Screen) transitioning, hence screen classes only needs to hold this interface instead the whole [JeuDeFootball](#_JeuDeFootball) object.

### WorldContactListener

This is Box2D’s ContactListener implementation, which modifies behaviour depending on physical interactions. For this project, only the beginContact function is used. This’ll give us the two Fixtures participating in a collision. We check if both has a userData, otherwise we don’t have anything to do with this. If one of the fixtures is a [**BALL**](#_EntityTypes) we could do one of the following:

* The other fixture is [**LEFT\_GOAL**](#_EntityTypes)**:** a score is pointed for the right player
* The other fixture is [**RIGHT\_GOAL**](#_EntityTypes)**:** a score is pointed for the left player
* The other fixture is [**PLAYER**](#_EntityTypes): shoot sound is played

## Game Logic

The Game class is where most of the logic is implemented.

### Fields

* **List<**[**Entity**](#_Entity)**>** – The list of the entities in the game
* **World** – The Box2D World
* **float timeLeft** – The time left in the game
* **float accumulator** – This variable is to accommodate to lagspikes, and be able to play on slower devices
* **Score** – The current score of the game
* **Boolean resetPositions** – This is used to indicate the game logic that it should reset the position of the entities.
* **GlyphLayout** – is used for determining the width of a given text
* **float padSize** – The size of the controllers
* **Vector2 leftPad, rightPad** – The center location of the controllers
* **TransitionListener** – an instance for [TransitionListener](#_TransitionListener) to facilitate screen changes
* **float goalTimer** – The remaining time from a goalScreen (starting at 1 second)
* **float gameOverTimer** – The remaining time in case the time is over (starting at 5 seconds)
* **Boolean singlePlayer** – Designates whether this is a Single Player game
* [**Button**](#_Button) **exitButton** – The exitButton instance

### Constructor and Init

The constructor only gets one parameter, whether this is a singlePlayer game or not. After that, the fields will be initialized, the timer is set to 180f, the ContactListener of the World is set to a new instance of [WorldContactListener](#_WorldContactListener), the exit button is also added.

We populate the entities array as follows:

* We add the [Ball](#_Ball) to the center location
* We add the [LeftPlayer](#_LeftPlayer_and_RightPlayer), with the [controls](#_PlayerController) [W,A,S,D], and the pad location on the left, with a radius: padSize \* 1.5
* We add the [RightPlayer](#_LeftPlayer_and_RightPlayer):
  + If it’s a Single Player Game, we set the controller to an [AI controller](#_CPUController), targeting the left pad location, as it’s next to the goal
  + If it’s a Two Player Game, we set the [controls](#_PlayerController) [UP, DOWN, LEFT, RIGHT], and the pad location on the right, with a radius: padSize \* 1.5

We calculate the aspect ratio of the playing field (needed for scaling the borders of the map) and set the scale to 1/100th of the height, as it’s used for calculating the onscreen position from the physical location and vice-versa. After it’s set, we could instantiate the physical Bodies for the entities.

We load the border vertices from the model file, scaled by the aspect, and we create the edge of the map with EdgeShapes, which are StaticBodies with a restitution of 1.

We also add the goals to the World, these are rectangles with isSensor set to true, so they won’t make collisions with other objects, and we set their userData to [**LEFT\_GOAL**](#_EntityTypes) and [**RIGHT\_GOAL**](#_EntityTypes).

### Score

The score(Boolean left) function will increment the left or right player’s score by one, play the *goal* sound, and set the goalTimer to 1

### GlyphWidth

This function gets a String as a parameter, and gives back the width of that text rendered with the game’s font, used for layout purposes

### Tick

This function is implemented from [Screen](#_Screen), this will be called from [JeuDeFootball](#_JeuDeFootball) to facilitate [Entity](#_Entity) updates. The workflow is the following:

* Check if resetPositions is true. If it is, reset the entity positions, and set resetPositions to false
* Check if exitButton is pressed. If yes, execute it’s onClick() method, which will return us to the [main menu](#_Menu).
* Set frameTime to min(delta, 0.25) to avoid lag spirals
* If timeLeft > 0, we are in playing mode, and we should do the following:
  + If goalTimer greater than 0, we are showing the goalScreen, so we only decrement that with the frameTime, and when we reached 0, we should set resetPositions to true. We shouldn’t do anything else, so we return from the tick function.
  + Decrement timeLeft with frameTime
  + Add frameTime to accumulator
  + While accumulator is greater than STEP\_TIME, we are stepping the physics for STEP\_TIME
  + Go over the entities, and call their *tick()* method.
  + If timeLeft reached 0, set gameOverTimer to 5 seconds
* If gameOverTimer is greater than zero, we are still showing the Game Over screen
  + Decrement gameOverTimer with frameTime
  + If gameOverTime reached 0:
    - Set resetPositions to true
    - Reset the score
    - Set timeLeft to 180 seconds

Remarks:

We should separate resetPositions because otherwise it may reset the Body locations while in the step function, and that’d lead us to an assertion failed error.

### Draw

This is also implemented from [Screen](#_Screen), this function is responsible for drawing the playfield. The workflow is the following:

* Draw the background image
* Call the draw function of the [entities](#_Entity)
* Draw the left pad with yellow color
* If it’s not a Single Player Game, draw the right pad with cyan color
* Draw the scoring and the time left on the top of the screen
* If goalTimer is > 0, draw the goal overlay on the screen
* If gameOverTimer > 0, draw the “GAME OVER” text, and the winner on the screen