Mestrado Integrado em Engenharia Informática e Computação

Laboratório de Programação Orientada a Objetos

eLMaze

Final Project Intermediate Check-Point

Filipa Manita Santos Durão, up201606640@fe.up.pt
Rui Pedro Moutinho Moreira Alves, up201606746@fe.up.pt

28-04-2018

Project's Description

Our project, **eLMaze**, is a single/multiplayer game for desktop **or** for desktop and mobile, using mobile as a client remote controller and desktop as a server.

The objective of the game is to move a ball through a serie of mazes (levels), reaching an exit to proceed to the next level.

The game is composed by two main game modes:

- **Single Player**: A player can play using the mobile as a remote controller, using the device's accelerometer to move the ball through the level. Alternatively, the player can also play in the desktop only using the arrow keys.
- **Multi Player**: The multiplayer mode allows two players to play the game, working together to solve the levels. They may both connect with their mobiles, moving their own ball as described before. Alternatively, they may also play in the desktop only, using the arrow keys and the W-A-S-D keys.

As for the secondary objectives, this project will feature:

- **Networking**: The desktop acts as a server where the game is displayed and the clients (mobile app) connect to the server to play the game. The communication is established using native java sockets API, and is bi-directional
- **Physics**: The game runs on a libGDX physics engine, having collision detection to decide game's logics events (finishing levels, openings doors, colliding with walls, …)
- **Mobile:** The mobile client app uses functionalities such as the accelerometer to control the ball's movement and touch screen to advance through the game's menus.

Architecture Design

The package and class UML diagrams can be found in **desktopUML.png** (desktop project) and in **mobileUML.png** (mobile app project).

Although, we would like to add some additional notes regarding classes responsibilities and structure. Regarding the desktop project:

- We have adopted a full **MVC** (Model View Controller) architecture, with two additional modules, one for networking and other for unit testing.

- **Model:** This package contains all the game objects and levels models. The class **GameModel** is responsible for managing all the models creation and initialization. A **LevelModel** (abstract class) can be a **SinglePlayerLevelModel** or a **MultiPlayerLevelModel** (both abstract classes) that can be extended to create any level model. All game objects are represented by the **EntityModel** abstract class, that is extended by all the game object models (**BallModel, WallModel, …**)

- **Controller**: This package controls the game flow and contains the game's physical world representation. The class **GameController** is responsible for managing the game flow (Level Changing, Winning condition, …). It is associated with a **CollisionListener** class object, that is responsible for managing all the collisions within the game. It contains a list of **Level**s, that can be either **SinglePlayerLevel** (containing a **SinglePlayerLevelModel**) or a **MultiPlayerLevel** (containing a **MultiPlayerLevelModel**). All the levels contain their own physical world, containing objects of the class **EntityBody**, representing all the world's physical bodies. This class is extended by all the game's bodies (**WallBody** containing its own **WallModel**, **BallBody** containing its own **BallModel**, …).

- **View**: This package is responsible for managing all the GUI and all the game visualization. It contains a set of menus (**MenuView**, **InstructionsView**, **GameView, …**). The **GameView** class contains a set of all the **EntityView** objects (created by the class **ViewFactory**, responsible for creating EntityViews), that represent the visual representation of all the game's objects and is extended by all the other game objects views (**BallView** containing its own **BallModel**, **ButtonView** containing its own **ButtonModel**, …)

- **Network**: This package is responsible for setting up the server socket and managing connections, broadcasting and receiving messages. The **NetworkManager** class is responsible for creating the server socket and for creating a **SocketManager**, on a separate *thread*, that is responsible for waiting for client connections and managing them, creating a new **SocketListener** whenever a new client connects (up to 2 clients, depending on the game mode), also on a separate *thread*, that is responsible for receiving (**MessageToServer** class objects) from the client and sending

messages (**MessageToClient** class objects) to the client specified by the given socket. This package uses the native java sockets API.

- **Test**: This package is responsible for performing unit tests in the game, using the java JUnit4 API.

Regarding the Mobile App Project:

- **View**: This package is responsible for managing all the GUI and all the game app visualization. It contains a set of menus (extended instances of the **MenuView** class, which are created by the class **MenuFactory**, such as **ServerConnectionView**, **PlayGameView, …**). This package uses the libGDX stage and actor abstractions. The classes **ImageFactory** and **ButtonFactory** are responsible for creating **Button** and **Image** actors for the scene.

- **Controller**: This package is responsible for managing the app's logic, making a "bridge" between the app's View and the Networks.

- **Networking**: This package is responsible for creating a socket and connecting to the server socket, receiving (**MessageToClient** class objects) and sending messages (**MessageToServer** class objects) from and to the server (desktop). This package uses the native java sockets API.

- **Test**: This package is responsible for performing unit tests in the app, using the java JUnit4 API.

Regarding the Design Patterns, the ones that are expected to be used are the following:

- **MVC - Model View Controller**: To keep the game's view, model and logics not dependant to each other, achieving a more modular design.
- **Singleton**: To ensure that a given set of classes have one and only one instance to be accessed (GameController, GameModel, NetworkManager, …)
- **Observer**: In the controller package, the CollisionListener will act as a collision observer for the game's physical world (this DP is implemented by the libGDX library)
- **Factory**: To keep the responsibility of creating objects to a specific class (In the Mobile app Project, MenuFactory, ButtonFactory and ImageFactory classes from the View package)
- **Factory Method**: Used in desktop's EntityView class to make object instantiation defer from each of the EntityView's subclasses.
- **Flyweight**: Used in desktop's ViewFacotory class to keep a cache of views and re-using them in the future.
- **State**: Both mobile and desktop applications contain their own state and have events that change that state.

GUI Mockups

Desktop Mockups

As mentioned above, the Desktop will have several **menus** (views) for the player(s) to choose which game mode it / they intent to play.

The first view is the **Main Menu**, the beginning view when the game is launched. In this view, the player can either choose to see the **credits**, the **instructions**, to **exit** the game or to move on to **play** the game.

Image 1 - Desktop Main Menu Mockup

Considering the player chooses the **Play** option, the application will then move on to the next screen, the **Number of Players** choice screen. In this menu, the user will choose between playing in single player mode or multiplayer mode.

Image 2 - Desktop Game Mode Menu Mockup

After the Game Mode choice, the app will proceed to the next screen, the **Platform choice** screen. In this menu, the player will have to choose between playing the game using the phone as a remote control or using the keys of the computer's keyboard.

Image 3 - Desktop Platform Choice Menu Mockup

Finally, the last menu before starting playing the actual game, is the skin choice menu and the presentation of the **Game Code**. In this menu the player(s) is asked to choose the aspect of the ball with which it will play. Besides that, the Game Code presented has to be inserted in the player's phone in order for them to be able to connect to the server. This menu view will depend on the choice of the Game Mode (single or multiplayer).

Image 4 - Desktop Single-Player Pre-Game Menu Mockup

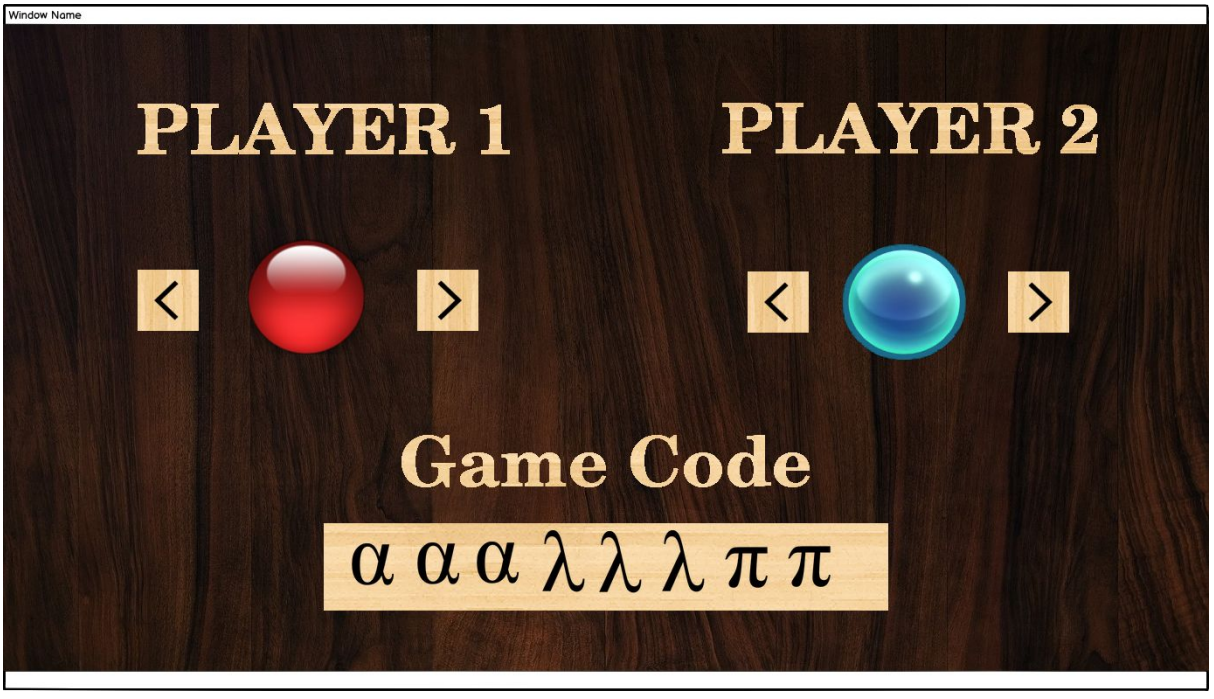

Image 5 - Desktop MultiPlayer Pre-Game Menu Mockup

Android Mockups

The Android app for our game will also have several menus, similarly to the Desktop application.

The app starts by its **Main Menu**. In this menu, 4 buttons will be presented to the user: **Play**, **Instructions**, **Credits** and **exit**.

Image 6 - Mobile App Main Menu

The **Play** button will lead the player to the **Game Code** view. In this Menu, the Player must introduce the code that is presented on the Desktop's Screen in order to start the game.

Image 7 - The Mobile App Game Code view

Once the Game starts, the user can start playing by tilting its phone. The following screen will be presented at this stage.

Image 8 - The Mobile App Play Screen

The **Instructions** and **Credits** views will be very similar, only a text table showing the necessary information and an exit button to go back to the main menu.

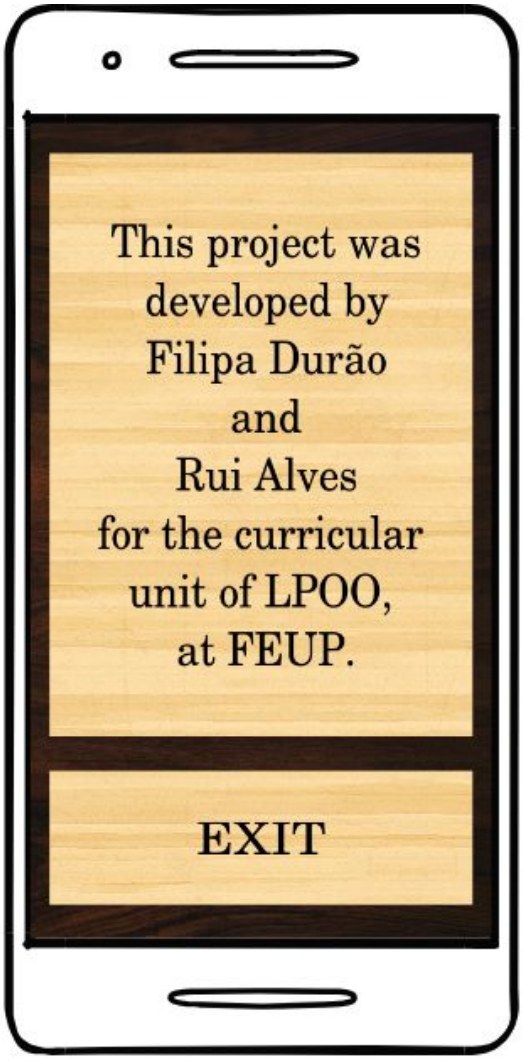

Image 9 - Mobile App credits page Mockup.
The Instructions view will be similar, only changing the text content.

Test Design

Regarding the Desktop project, tests will feature:

- **Entity Models** testing: Every model that represents a game object entity will have their attributes and methods tested (constructors, position getters and setters, radius getters and setters, width and height getters and setters, and other more specific methods, such as clicking methods for buttons, opening and closing methods for doors, …)
- **Level Models** testing: Level methods constructors, get and set methods.
- **Network messages** testing: Network Message classes will also be tested (constructors, get and set methods and other specific methods)
- **Network utilities** testing: Utilities such as IP parsing methods.

Regarding the Mobile app project, tests will feature:

- **Network messages** testing: Network Message classes constructors, get and set methods and other specific methods will be tested.
- **Network utilities** testing: Utilities such as IP parsing methods.