Architecture

Team 10: Hard G For GIFs

Dragos Stoican

Rhys Milling

Samuel Plane

Quentin Rothman

Bowen Lyu

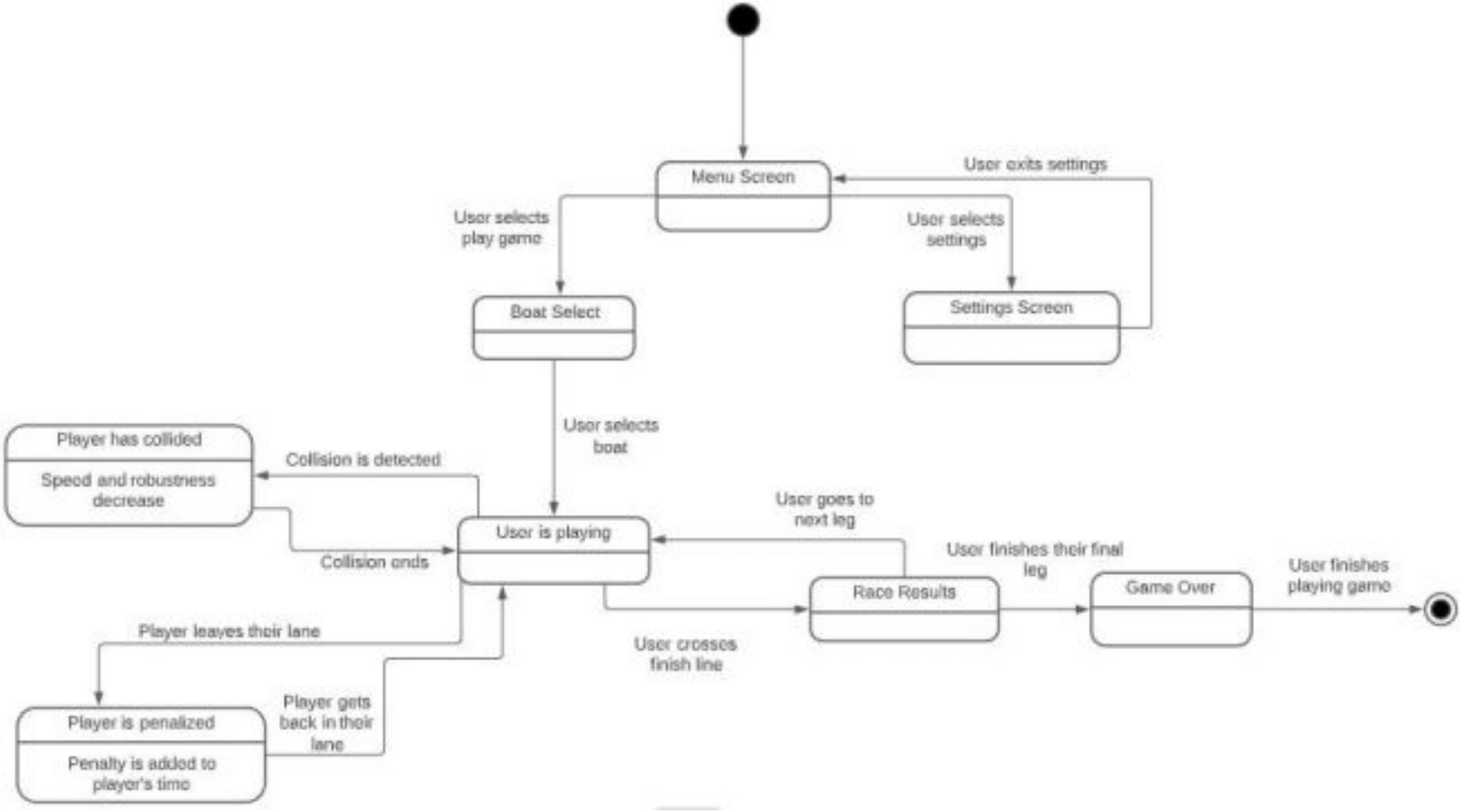
Jack Gerhard

a)

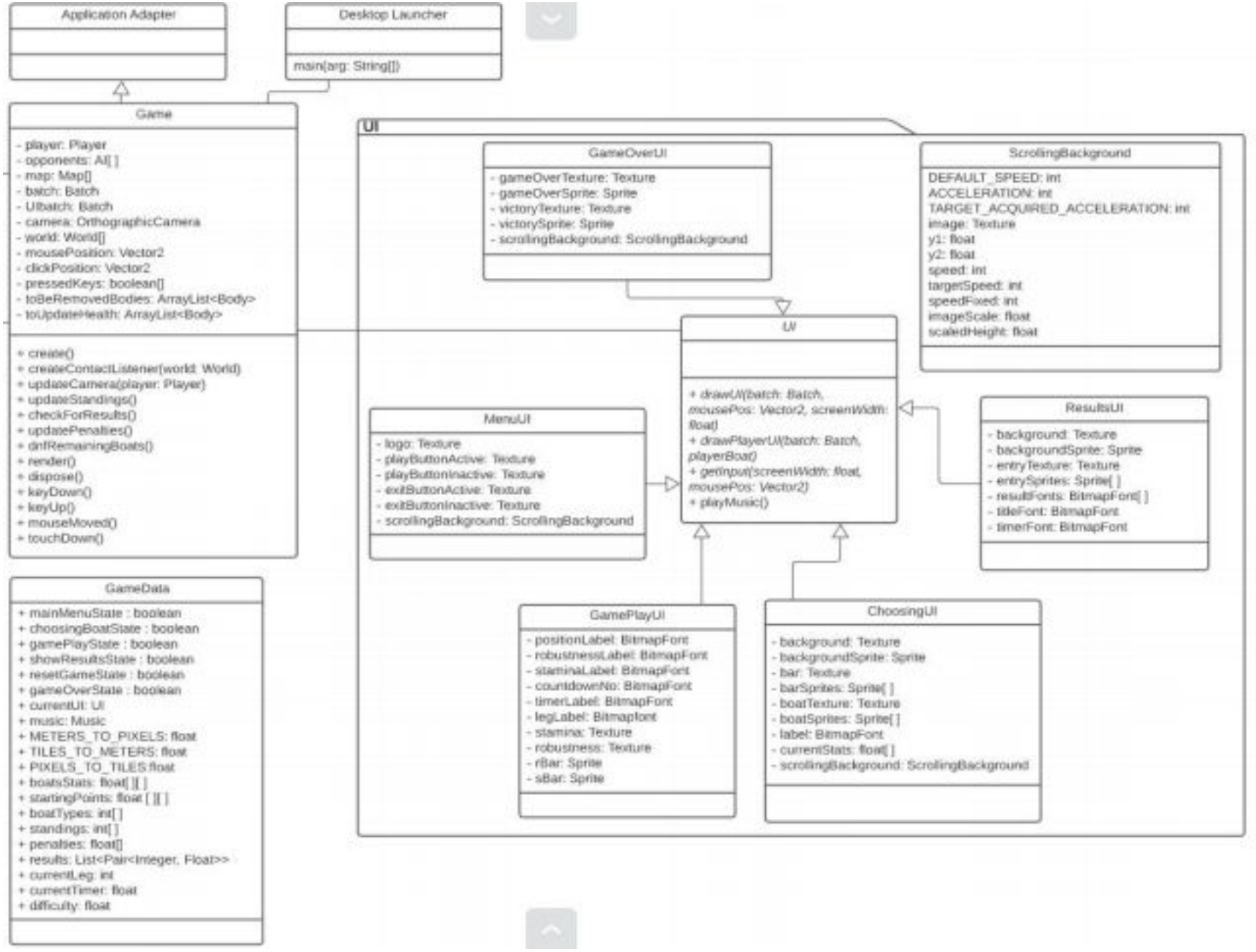
- We used Lucidchart to represent the two architectures.
- We created a UML State diagram for the Abstract architecture.
- We designed a UML Class Diagram for the Concrete architecture, using an OOP Entity Hierarchy with inheritance and

polymorphism.

Abstract architecture diagram



Concrete architecture diagram



b) Abstract architecture:

- The abstract architecture represents all the states that the users go through during the game and the actions that transition them from one of those states to another. Some of these states also describe the result of entering them, like colliding with an object or going out of the lane.
- The abstract architecture details the player's progression from the start of the game, selecting the boat and competing, reviewing their results, and finally ending the race day once they have finished all the legs they compete in. ● We used a state diagram to show everything the game has to do, then divide these requirements into smaller tasks. ● Using a state diagram also makes it easy for us to construct the concrete architecture, as it is a more refined version of the abstract one, where we go into detail about implementing the small tasks that compose those states and the transitions. Concrete architecture:
- The concrete architecture adopts an inheritance-based style, focused on easy implementation of the abstract architecture ● Our project is divided into 3 main parts, represented by the two packages we created, and the main game class. Additionally, we use a static class to store information about the game that is necessary for all the other sections. GameData class
- The static class, GameData stores the data needed for the game, including the current state of the game, the UI drawn on the screen, the difficulty of the game (UR_DIFFICULTY), hardcoded specs of each boat type, and the player's boat decision ● On top of that, as we use Box2D to handle the game's physics, and Tiled to create the maps, all of which use a different unit of measurement, GameData stores the ratios between pixels, meters, and tiles as necessary.

Game class

- The main game class, called Game, handles the creation of the world physics and the maps, this is also where we update objects after collisions, the boat standings, and results. (UR/FR_DAMAGE, UR_LANE, FR_PENALTY) ● Here we decide what is rendered on the screen, based on the game state found in the GameData class, implementing movement through all the states as described by the abstract architecture.
- The main class also handles the player's input, passing those parameters to the other methods.

Core Package

- The Boat class stores all the specs, handles the movement, rotation, physical properties, and rendering of each particular boat. We also include here the ties of the boat with the lane it is located in. (UR/FR_CONTROLS, UR_CHOOSING_BOATS, FR_STAMINA, FR_STATS, FR_ASPECT, FR_VARIABLE_CONTROLS)
- All the race contestants are an instance of a boat, thus we can control both the player and the AI by re-using methods with different parameters. (UR/FR_CONTROLS)
- The Player and the AI class extend Boat and add to it a updatePlayer or updateAI method which call the inherited moveBoat and rotateBoat methods, which are necessary for the gameplay state of the game
- The Map object is used to create an instance of a map. This includes the necessary functions for creating physical objects in the world, the lane boundaries, and the finish line. (UR_MAP)

● Every map object has an array of Lanes which is created in the createLanes method. Every lane is filled with obstacles and its boundaries are created and made available for the boats to use. This allows for the detection of collisions with obstacles and passing outside the lane as required by the abstract architecture. (UR/FR_LANE, FR_OBSTACLES) ● When creating an obstacle, we randomise its type and create its body using the Obstacle class methods. (FR_OBSTACLES)

UI package

- The abstract UI class is used as the declared type of a variable which will store the current UI to be displayed. It is instantiated with its subclasses so that each state of UI can be implemented in a separate class.
- The UI class has a getInput method to respond to user input. The method is declared here as several sub-classes will need to implement it, each in a different manner based on what game state they are displaying.
- UI declares two abstract methods, drawUI is responsible for drawing static components to the screen, such as the menu, whilst drawPlayerUI is responsible for player-related elements, such as the bar representing the player's stamina in a race. ● The playMusic method is called by all the inherited classes to play the current soundtrack from the GameData class. ● MenuUI and ChoosingUI display static elements that transition the user from the start of the game to the beginning of the gameplay, choosing a boat in the process and saving it in the GameData class (UR_MENU, UR_CHOOSING_BOATS) ● GamePlayUI is displayed during races, showing the player's remaining robustness and stamina, their current time in the leg, and their position in the race. As this UI is constantly changing, we use the drawPlayerUI method, allowing us to access the current stats of the player's boat. (UR_HUD, FR_HEALTHBAR)
- ResultsUI displays the results of each leg, showing each team's position and the time they took to complete the leg. ● GameOverUI is displayed when the player finishes their legs, showing a victory screen if the player won, or a game over screen otherwise.