Implementation

Team CASEUS

(Team 7)

Eliot Sheehan

Matthew Turner

Daniel Atkinson

Hannah Pope

Mayan Lamont

Divyansh Pandey

Implement support for different levels of difficulty

There was already difficulty scaling as the races progressed based on an integer, we took advantage of this to implement the requirement UR_DIFFICULTY_SELECTION. This was done by setting the initial difficulty to a value based on the player's difficulty selection then increasing that value as each race passes instead of having each race a predetermined difficulty. We had to update the MenuScreen class to show the different difficulty options and write code so that it would detect which option was pressed. The DragonBoatGame and GameScreen classes were edited so that they would run the correct legs of the race (initial races versus the final race) and show the correct version of the leaderboard (highlighting the top 3 boats on the second-last leg).

Implement five power-up packs

To implement the power-up packs in accordance with UR_BOOST_ORBS we created the Boost class which extends the obstacle class. Even though boosts aid the player unlike the other obstacles, we thought that inheriting the Obstacle class was the best method to take since the collision checking and movement was already implemented and working for the Obstacle class. From there we had to find a way to apply the boosts to the boats. This was done by changing the relevant attribute of the boat depending on which type of boost it hit and starting a timer; once the timer finishes the boost effect is removed by reverting the attribute to its original value. The last changes we had to make was to update the GameScreen and Lane classes to spawn the boosts, this was done by simply extending the current obstacle spawning functionality to include the new class.

Implement saving and loading

The approach we took to saving and loading for the requirements UR_PAUSE_MENU and UR_LOAD was to save the variables needed to recreate the objects in JSON form then when loading, the data from the JSON would be extracted and the objects rebuilt. To minimise the complexity of saving we first analysed which classes actually needed to be saved and from them we found which of their variables needed to be saved. For some classes we intentionally omitted variables that they need from the save because these variables are referenced from multiple classes so would cause redundant data in the save as well as issues when loading; instead these variables are saved by 1 class and passed to the others when they are recreated.

Each object that will be saved has its own function "toJSON()" which aggregates its variables that must be saved into a HashMap then returns that HashMap as a JSON string. The objects also have their own function to rebuild themselves, "make*classname*", which returns a new object of the same type and sets its attributes with the data given to it.

Writing saves to file, loading them back and converting data between JSON and Java data structures such as HashMaps is handled by the IO class.

To allow the player to make use of saving their game we added a pause menu in the GameScreen class, which gave the options: "Resume", "Save and exit" and "Exit without saving". When this menu is opened the game is paused to allow the user to select their option without panicking. To load the game we added a button to the bottom of the menu screen in the

MenuScreen class which, when clicked, will load the player's save game unless the save game doesn't exist in which case it will notify the player. The saved game is started in a paused state to allow the user to regain their bearings on what the state of the game was when they left it.

Implement boat death

To complete the requirement from assessment 1, UR_LOSS, we had to make the player lose when they die. To do this we added some code in the GameScreen class to check if the player's durability is 0 or below, if it is the method to end the game is called. In the DragonBoatGame class, now that the player could die, we added a check on the player's durability when ending the game to prevent them from getting a medal if their durability is 0 or below.

End race when the player finishes

To improve on the requirement UR_TIME first we decided on an algorithm to use to predict the boats' finish times. We settled on a linear algorithm which would predict the finish time based on how long it took the boat to travel the distance it has. We chose this because it is a simple algorithm so would not take long to implement as well as having an insignificant performance impact, but also because we thought it was fairly accurate. Next we modified the end condition for each race to end when the player finishes and predict the times for the remaining boats. These changes only affected the GameScreen class.

We felt this improvement would be worthwhile because it has no effects on any parts of the code external to the GameScreen class and potentially has a considerable improvement to the player's satisfaction with the game because they won't get bored waiting for the opponents' boats to finish.