Assessment: 2

Deliverable: Implementation

Team Name: Team 8

Team members: Charlie Hayes, Matilda Garcia, Joshua Stafford, David Kayode, Ionut Manasia, Matthew Tomlinson

Report of New Implementation (Impl b)

The final version of our implementation implements both our updated architecture and requirements. Specifically, areas that did not require changes were kept as such to closely align to the original concrete architecture and requirements outlined. Then, to meet the changes made for assessment 2, additional classes and methods were implemented in accordance with the updated architecture to fulfill newly elicited requirements such as UR_SAVE, along with some rewriting of portions of code.

The requirements referenced can be found here: Req2.pdf (eng1-team-8.github.io)
The architecture referenced can be found here: Arch2.pdf (eng1-team-8.github.io)

The significant new features and the changes to implement them are as follows:

Saving/Loading

This feature allows the player to press the escape key during gameplay to save the game and exit to the main menu. Saving is implemented via two static methods in the 'SaveLoad' class, as shown in the concrete architecture with the 'save()' method being called in the Game class and 'load()' being called in the 'MenuUI' class. The original intention was to save the exact game state (progress through the leg, obstacle placements...), however, due to time constraints, this feature was cut down to just save the most relevant game information.

Saving works by using LibGDX preferences to create persistent storage for the current leg, the player's boat and difficulty choice and the best times for the player and opponents by serialising them to json.

Loading essentially performs the reverse process, retrieving the stored serialised items, converting them into the appropriate java object/data type and setting the relevant variables with their values. When the player selects the load option on the main menu, the 'load()' method is called and the game is rendered with the saved attributes, loading the player to the start of the leg they saved on.

This new feature was implemented to fulfill the requirement UR_SAVE and all changes for this feature were new additions so there are no side-effects of its implementation and does not extend from any other classes.

Powerups

This new feature required the most new additions and changes. Most notably the new class Powerup was added to represent a given powerup. 5 powerups were added, each using a unique sprite that applies an effect onto the colliding boat. A decision was made to extend Powerup from Obstacle as for all intents and purposes they are obstacles (except they produce positive effects on collision). Therefore, they could be handled just like objects with the same physics and collision detection methods. The Powerup class has an additional attribute, type, which is an integer which can be parsed to determine which effect to apply. Also, a modified createObstacleBody() method was implemented to use the new createBox()

method which allowed us to make the collision body a simple box rather than using the BodyEditorLoader from the previous software.

A side effect of extending from Obstacle, however, is that the game recognises them as obstacles as well as power ups, meaning that a check for a collision with an obstacle will also include collisions with power ups.The implementation can be justified though as the side effect was easily remedied by checking for Powerup collisions with a Boat first through selection and then generalising to Obstacle collisions with other objects. These collision checks were implemented by modifying the pre-existing Box2D contact listener, adding the Powerup to the toBeRemovedBodies list and calling a newly implemented Boat method applyPowerup().

The Boat class was changed to have a new function applyPowerup() with the powerup type as a parameter. The function uses a switch statement to determine which of the 5 new Boat methods should be applied based on the passed powerup type. The other 5 new methods are: regenRobustness() which adds 20% of the boat's robustness back, up to its starting amount; staminaBoost() which adds 20% of the boat's starting stamina back, up to its maximum; speedBoost() which increases the boat's maximum speed value by 5.0; accelerationBoost() which increases the boat's acceleration value by 5.0 and maneuverabilityBoost() which increases the boat's maneuverability value by 5.0.

Finally, a new method was added to the Lane class which uses a modified version of the obstacle spawning algorithm in conjunction with a new Lane attribute, a powerup array. Its purpose is to evenly, but randomly, populate the map (and powerups array of set length) with powerups which have a randomised type. Consideration was given to altering the AI to steer into power ups, however, we found that the AI was already collecting power ups with a sufficient frequency whilst dodging obstacles.

All of these changes were made to fulfill the requirement UR_POWERUPS, with 5 powerup types being added, each having a unique sprite and altering the given Boat's attributes accordingly.

Difficulty selection

This new feature allows the player to select a difficulty from the main menu. The choices for difficulty are 'easy', 'normal', and 'hard'. This feature works by using the pre-existing MenuUI class and GameData.difficulty data, not requiring any additional classes or methods. Originally, the difficulty information was used to increase the AIs capabilities as the legs went on. Therefore, MenuUI could be changed to set a difficulty float array depending on the difficulty selected. This can be justified as it fulfills all the specificities of the customer's requirement and also keeps the increasing leg difficulty.
This implemented AI difficulty adjustments, however, in accordance with the requirement FR_DIFFICULTY the difficulty had to increase the number of obstacles being spawned. The obstacle generation process was altered to set the length of the obstacle array to be multiplied by the difficulty level for the given leg.

This change meant that obstacles could not be spawned before the player had selected a difficulty. To accommodate this, obstacle generation was moved into the render() method,

inside of an if clause, meaning that obstacles are now spawned as a leg starts (immediately after the boat selection). This has the side-effect of introducing a brief pause before each leg whilst the game generates all the obstacles, but ultimately was the most sensible way to approach this issue as regardless, a slow-down will need to be introduced at some point after initial game creation.

The changes made that were not new, but required to complete the product:

Competition structure

This change was needed to fulfill the requirement UR_LEGS and to better match the competition structure laid out in the product brief. The new structure consists of 4 legs, the first being a practice and the last being the final, the top 3 best leg times progressing through to the final. To complete this change a new array storing best times was added to the GameData class, with a best time being overwritten if the finishing time is less than the current best time. Further modifications were made to the GamePlayUI class to display “Practice” or “Final” and to ResultsUI to display the best times so far.

Additionally, a new algorithm was implemented within the render function which determines which boat has the slowest best time. If the boat belongs to the player, the GameOverUI class is invoked to end the game due to DNQ. If the boat belongs to an opponent, the boat body is destroyed and is given a DNF time to prevent them from participating in the final. Now that the player could fail to qualify through timing, the decision was made to remove sections of code that game-overed the player if they broke their boat or came last place in a leg, instead opting to use the new ‘best times’ system.

Performance optimisations

A large part of the Game class’ create() function was rewritten to no longer generate all legs as separate objects with unique physics worlds at startup. Instead, the game now uses a single physics world and objects within the game are reset after each leg rather than iterating through an array of unique game worlds.This change was needed to separate the single large ‘loading’ time, when a difficulty is selected, into shorter pauses before the start of each leg and, as a positive side-effect, reduces the complexity of the game code.

A new function within the Game class called reset() was needed and can be seen in the concrete architecture of the system. This new function facilitates the reinstantiation of many game elements to create a ‘fresh’ leg. The function also has the ability to check if the player has reached the end of the game and will therefore perform a full reset and return the player back to the main menu, as if they had just started up the game. By implementing these changes, the requirement NFR_FAST_TRANSITION can be better met and risk T2 is less likely to occur.

Minor extensions

The minor extensions on their own do not form a significant change, however, when grouped together they do have an affect on the product: Graphical updates were made such as an updated water appearance to remedy graphical glitches; Obstacle physics were updated to make them dynamic objects that move (in createObstacleBody() within the class Obstacle) ;

On-screen controls were added to the gameplay UI (within GamePlayUI class); “Click to continue…” on UI screens to direct players and descriptive game-over screens (within GameOverUI).

The more descriptive game-over screens were a slightly larger change as they incorporated checks to determine how the player finished the game, but it was mostly a case of updated assets.

These changes were needed to to fulfill the requirement UR_DISPLAY_INSTRUCTIONS and complete the overall product to a consistent visual standard (CR_GRAPHICS_CONSISTENCY).

Features Not (Fully) Implemented

The extended implementation covers all requirements elicited to an extent. However, the implementation of some of the requirements is not fully complete and may not be satisfactory for the customer. Instances of this are listed below:

UR_SAVE 1.3.3

The implementation of saving included in the current version of the game is rather rudimentary but does allow the player to quit and resume to a part of the game they left at. The customer specified that they wanted a full game state save and resume, wherein the player (and opponents, obstacles, power ups etc.) exact positions would be saved and resumable. However, due to time constraints, this level of saving was not feasible and instead the decision to implement a more attainable approach was taken. This approach is allowing the player to save at any point during gameplay and load to the start of the leg they were on, with their chosen boat/difficulty etc. This is close to what the customer requested but not a full implementation.

UR_DISPLAY_INSTRUCTIONS 1.2.2

New user interface elements instructing the player how to continue, control the boat and save etc. were added, however, full instructions on how the game is structured and win/lose conditions are not explained in the game. We felt this was an acceptable requirement to not completely fulfill as a manual for the game is now included on the game’s website which explains how the game works.

NFR_FAST_TRANSITION 3.0.1

Despite major changes made to improve the state of the transition from menus to game play, the groundwork laid for the generation of obstacles as physics bodies is inherently more intensive than it ideally would be. A complete rewrite of this aspect was outside of the scope of time allowed, however improvements were made as shown in the systematic report. The delay just before gameplay starts is very small but does exceed the fit criterion laid out in the requirements documentation on lower end machines. Sufficiently powerful machines will complete the transition within the acceptable time, however, and all other transitions are acceptable too.