Part 3. Implementation and Changes

Team 5 | Team Pending

Thom Robinson
Iris Scoffin
Ayman Elsayed
Annice Headman
Izaac Threlfall
Yihong Zhao

Part B - Changes and Reasoning

In this section we briefly explain the six major changes to the game we had to make in order for the game to satisfy the new requirements we received in assessment 2.

Change #1 - Combat with enemy ships

When we inherited the game from the previous team, we acknowledged that the enemy ships had no collision with the player, bullets or other enemies. Firstly, the issue with bullet collision was solved by changing the class Enemy to implement the interface IHittable as well as adding any enemy created to the ArrayList “hittables” in Game. This change ensures that any bullet will check for collisions with instances of the Enemy class in the class Projectile’s update() function, calling the enemy’s hit() function, which handles damage and checking whether the ship should be destroyed. Similarly, when movement is calculated in Enemy’s update() function it will check if it would overlap with the player ship and vice versa, creating a collision if this happens.

If an enemy ship is destroyed it rewards the player with a variable amount of gold (10-20) and XP (5-10), this decision was made to fit in with the variable way gold and XP are distributed upon destroying a college. As enemies can be destroyed and reward gold and XP, the functional requirement F_Ship_Death is satisfied.

In order to satisfy F_Ship_Attack, the enemy ship had to attack the player. To do this we implemented a two-state movement system such that whenever the player is not “in range” of the player, it makes random moves around the map. But when in range the enemy ship tries to move towards the player while firing cannonballs at them that deal damage. The use of a single range value makes it easy to tweak the “aggression” of enemy ships by altering a single value. Also, no changes were needed in the projectile class, as enemy cannonball logic exists in the Projectile class and is already used in the College class.
Finally, it was decided it would be appropriate to add health bars to the ship using code from the player ship class, this gives the player a rough idea as to how many shots they need to fire at the enemy. Furthermore, to make the enemy ship seem more natural, the particle effects for movement, smoke and damage from the Player class have been implemented into the Enemy class, making the game more graphically appealing.

Change #2 - Obstacles

Obstacles were a completely new feature that needed to be added to the game in order to satisfy the functional requirement F_Obstacles. We needed to make different obstacles on the map that would affect the player in different ways depending on the obstacles they hit. First of all, we made a new class called Obstacles with a constructor that has the game, the position of the obstacle and the type of obstacle as the arguments.

We added three different obstacles to the game, those being rocks, sea mines and icebergs. Each obstacle affects the player in a different way and changes the way the player approaches the level. We then decided to code each obstacle individually with its position,

that way we can change the layout of the obstacles as we see fit and not to make the game unfair for the player.

The effects of each of the obstacles are as follow:

- Rocks: Slows down the player and damages the player for each frame the player is inside the rocks
- Seamine: Explodes, dealing a considerable amount of damage to the player
- Iceberg: Breaks, dealing a small amount of damage and halting the player's momentum

These obstacles were laid out in a way that will make the player find it more difficult to manoeuvre the terrain and sometimes decide which is the best path to take to minimise damage taken.

Change #3 - Weather

Weather was another completely new feature that needed to be added to the game in order to satisfy the functional requirement F_Obstacles, specifically, relating to the part about adding weather events. At first we weren't sure how to go about implementing it, we then came to the conclusion that it will be mainly a visual effect that will make it harder for the player to complete their objectives, therefore adding an extra layer of difficulty to the game. The weather effects that were all drawn and are implemented as textures and sprites that cover the whole screen and follow the camera hence gives the feeling that the whole map is affected by the chosen weather condition.

There are two different weather conditions that we came up with.

- Rain: covers the screen with moving rain, making it more difficult for the player to focus on what's happening in the game.
- Fog: covers the screen with fog that gets darker towards the edge of the screen. This obscures the players vision making it harder to see into distance as well as making it harder to navigate the terrain in general.

We decided to link the occurrence of the weather effects to a condition, that condition being colleges being destroyed. After the player destroys any college there will be a 50% chance of a weather condition happening, the weather condition will take effect for only 15 seconds. The specific weather condition is chosen completely by random as well, which adds to the replayability of the game and makes it more exciting. Additionally, we added a UI timer that will specify which condition is taking effect and how long is left till it's gone, this helps satisfy the requirement NF_Stats.

Change #4 - Pickups and Purchasable Upgrades

To Implement the pickups and upgrade system we first had to create a way to change the player's stats cleanly. To this end we created the Buff class (a misnomer, as it can also apply debuffs). The Buff class stores a list of stats to change on the player, the amount it should be

changed by and the duration of the effect. The Buff itself also deals with the icon of the effect, selecting one based on the modified stats.

The Player class has a list of Buffs and getter methods for each stat. The getter methods query each buff in the list to see if it changes the stat, then returns the final value. As certain Buffs expire, the Player class had to be updated to reduce the time remaining. Furthermore, to show the time remaining for temporary Buffs we added extra UI code to display the icon of the buff and the time remaining.

To assist in the creation of Pickups and Upgrades, we implemented collectables. Collectables take the game object, their position as a Vector2 and the Buff to be given to the player. Collectables are rendered into the world, and have a floating animation made by offsetting the sprite by a function of a sine wave.

Pickups are an instance of the Collectables class and have all the same arguments. When the player collides with the pickup it adds its buff to the Player and then removes itself from the game. This satisfies the requirement F_Powers, providing temporary power-ups to the player.

Upgrades are also instances of Collectables, however, they also store the cost of the upgrade and render the stats that will be buffed and the cost of the upgrade. When the player moves over one and interacts with it by pressing “e”, if the player can afford the upgrade the cost will be deducted from the player object. This satisfies the requirement F_Plunder by allowing the player to spend their plunder they acquire from combat.

To add the pickups to the game world, we created a function called getRandomOverWater(), which chooses a random tile that is over water, and the final position of the pickup is randomised within that tile. To ensure separation between pickups, once a tile has been selected it is removed from the list of potential tiles, in contrast, the Collectables are added to fixed spots in the world to form a shop near the player's spawn.

Change #5 - Difficulty selection

In order to satisfy requirement F_Levels stating that players can choose their preferred difficulty level. This difficulty level would alter the game to provide a meaningful change in the game's difficulty to allow all users to access our game no matter their skill level. This helps satisfy requirements NF-Difficulty and NF-Audience, NF-Timing by ensuring that the difficulties account for our target audience of open-day visitors and ensuring that none of them find the game too difficult to complete in the given time (5-10 minutes).

One major change is to the UI of the start menu, we provided a message indicating the current difficulty level that could be altered using the arrow keys. To assist new players we included succinct tooltips stating the controls for changing difficulty as well as a description of how the difficulty affects gameplay, this helps support new users and satisfy the requirement NF-Help.

We provided four different difficulty settings (Easy, Medium, Hard and Impossible). These difficulties apply a percentage based change to the stats of colleges, enemies and the player

to alter difficulty. In particular we chose to change the health of entities as well as their firing rate, increasing the players health/fire rate by a percentage to make the game easier and reducing the same properties for the enemies/colleges, performing the opposite change to increase difficulty. Furthermore, the “impossible” setting disables the player’s health regen to add an extra challenge for those who want it. These percentage changes are simple to tweak and make it simple to change difficulty settings during testing or in development of future versions of the game.

These difficulty changes were tested to ensure that they did not interfere with the newly implemented upgrades and pickups and the additional buffs they provided the player, ensuring they did not interfere with the intended difficulty level too much.

Change #6 - Saving System

Finally, to satisfy the requirement F_Saving, we had to implement the functionality to save and load the game to an external file, allowing players to exit the game and restore their last game state later on. To do this two major functions were introduced, saveGame() and loadGame():

- The saveGame() function handles saving the game to an JSON document saved on the same file as the game called “save.txt”. The JSON document is formatted such that every type of game object has a JSON array in which all the objects of that type are stored as JSON objects consisting of every parameter necessary to rebuild that object back into its last state.
- loadGame() performs the opposite action and unpacks the JSON document and resets the game exactly as it was saved by reinstantiating all game objects with their exact properties (position, health, buffs etc.) and restoring game variables their previous state (e.g. current objective, time etc.).

To perform all JSON based packing and unpacking we use the JSON.simple library due to its easy to understand and use functions and good documentation. In particular JSONparser, simplified turning our save file from txt to a JSONobject that can be manipulated.

In order to allow the user to load and save the game we had to make changes to the existing UI. Firstly, we added an additional tool tip at the top of the screen stating users can press “esc” to save and exit the game. In order to load the game, we added a message on the start screen that would state whether a save file exists, and if it does, what date it was created, this message also prompts the user to press “enter” to load this save file and begin the game. However, if a save file does not exist then a message is displayed advising the user that they can make a save file at any point using “esc”. These tooltips help teach new players, and help satisfy the non-functional requirement NF-Help.

Our current saving functions use a fair amount of repeated code, with more time we would have considered investigating and producing a generic function for serialising and deserializing any given game object. However, we acknowledged this would not have impacted performance of loading and saving in the game, but it would have created a better standard of code.