Implementation

Group 12

Keira Lauder
Amy Mason
Dan Ho
Sina Khosravi
Yibing Wang

Implementation of Architecture and Requirements

Powerups

As part of requirement UR.POWER_UP and FR.DISPLAY_POWER_UP, where power ups need to be loaded onto the GameScreen when the player starts the game. We have created the class **PowerUps** which inherits all methods from its parent class **GameObject**. This follows the architecture that we have built [1][2]. The use of **PowerUps** is that it places the image of the powerup on the screen and once the player has hit the *hitbox* of the object, then it would *destroy()* the image on the screen so that it can no longer be used nor seen. Each power up is added to the screen through the **GameObject** class. It loads the texture and adds it into an array called *powerSprites*. When instantiating the **PowerUps** class here, the *powerSprites* parameter turns into the variable *frames*, it also gives the name of the power up in the string variable *powerup*. This is essential because in the *update* method, it finds the *powerup* and will call the method in the **Player** class to execute the power up's functionality.

This game has 5 different power ups: *GiveMoreDamage*, *HealthRestore*, *Immunity*, *TakeMoreDamage* and *Speed*. Each one of these has their own method in **Player**. For example, the power up *GiveMoreDamage* will call *screen.getPlayer().damageIncrease()* which will increase the amount of damage per **Projectile** that has been fired, but only for a set amount of time. This is the same principle for all the power ups.

Obstacles

As part of requirement UR.COLLISION and FR.COLLISION, we needed to have obstacles within the game to improve user experience. In this, we haved used Tiled [3] to add in some tiled images of rocks around the map. To enforce the collision, we edited the csv file which represented the map and replaced the number stored at the coordinate from -1 (which represented water) to 56, which resembled the obstacle. In the *create* method in the **YorkPirates** class, the previous team has already coded their collision. To do this, they have used the csv file and added rows where the cell value equals to -1. Within the **Player** class, the *update* method helps the user to move on the screen with the rows that have been predefined in the **YorkPirates** class.

Bad Weather

As part of requirement FR.BAD_WTHR, we needed to incorporate bad weather in the game. We wanted the user to sail over a section of the map which had bad weather (i.e. harsh waves) causing the player's speed to decrease. We have created a class **Weather** which inherits all the methods from its parent class **GameObject**, as shown in our architecture diagrams. This helps the image of the bad weather to appear on the screen. Like **PowerUps**, the bad weather is placed on **GameScreen** using a new texture which is stored in the *sprites* array. We have also used *weatherArray* which helps us to store the different bad weathers to be spawned on the screen. In the **Player** class, when the player hasn't sailed over the bad weather conditions, the *weatherMovement* is set to 1, meaning that the speed has not changed, but when the player does sail over the bad weather, the *weatherMovement* is set to 0.5, meaning that the speed of the player is halved.

Different Difficulty Types

With the requirement UR.DIFFICULTY, we needed to create different levels of difficulty for the user to choose from. We decided to have 3 difficulty types: Hard, Normal and Easy. In **TitleScreen** class, we removed the *startButton* variable that was there previously and replaced it with *easyButton*, *normalButton* and *hardButton*. The enter button was previously used to start the game, the functionality still works, but instead the game will be instantiated, and the *setNormal()* method will be called. In addition to this, we have also removed the listener for the *startButton* and instead have created 3 more listeners each for the different types of difficulty. In the same class, we have created a further 3 more methods to set the chosen difficulty level of the player that will feed to the **GameScreen** to make sure that the difficulty is met and then will alter variables in **Player**. Additionally, when enemies were added, the **GameScreen** also had the various enemy related variables changed. The methods **GameScreen**.*setEasy()*, **GameScreen.***setNormal()*, **GameScreen.***setHard(),* would call the **Player** methods of the same names, in order to manipulate the variables respectively. The GameScreen methods would change the enemy spawning variables: *enemySpawnFrequency*, *totalEnemiesAllowed*, *ambushRate*, *ambush_chance*, *ambushSize*. The Player methods would change player properties: regeneration, the damage the player towards the enemies, damage enemies dealt towards the player, maximum health.

Ways to Spend Plunder

With the requirement UR.SPEND and FR.SPEND, we needed to create a way for the player to spend their loot/money that they collect and earn in the game.We decided to have a shop where the player can purchase power ups. As shown in our concrete architecture, we have created a **ShopScreen** class which is a child class of **ScreenAdapter** that we have inherited from LibGDX. To have this new screen, when the player clicks on the pause button, i.e. the **PauseScreen** class has been called, where we have now created a further button to go to the shop. If the user clicks on one of the buttons in the shop (and they have enough plunder to buy it), the relevant power up variable **e.g. give_more_damage_bought** will be set to true. Within the method, *gameContinue()*, we determine which (if any) power ups the player bought, and then will call the power up method (which we have previously talked about in Powerups) that the player has bought. This is necessary as, if we just call the relevant power up method as soon as the player buys the power up, the timer for it will start before the player has returned to the game. The **ShopScreen** class creates a new screen which displays the power ups and has buttons to which the user would click on to purchase it. We have implemented listeners to do this. The *gameContinue()* method sets the screen back to the pause screen, ready for when the player wants to resume the game.

Saving the Game

With the requirement UR.SAVE and FR.SAVE, it was important to implement a saving and loading the game from where the player had previously left it. To do this we used libgdx preferences which creates a local folder .prefs containing a hash map which contains key values about the current game state. When the player loads the game back up it retrieves the values from the hash map and uses these to restore the game state. In the **PauseScreen** class, when the player clicks the ‘save & quit’ button, the *saveGame()* method is called which creates / overrides lots of keys and values in the YorkPirates prefs hashmap, such as game mode, the player’s position, their health, and the health and captured status of

the colleges. When the user clicks ‘load game save’ inside the **TitleScreen** class the *loadGame*() method is called if the hashmap contains values, loading the values and assigning them to the relevant variables, so that the player starts where they last saved the game. If the hashmap is empty (if the player hasn’t saved a game yet or if they won or lost the game in their previous game save) then nothing will happen if they click the ‘load game save’ button.

Combat with Other Ships

With the requirement FR.ENEMY_ATK, we needed to have combat with other boats. To achieve this we created the **Enemy** class. Within the **Enemy**.*update*() method, the enemy uses similar code to the **College** class in order to check when to shoot at the player. The Enemy class also uses a *SafeMove()* class similar to the **Player** class to check for collisions with the **GameScreen**.*tiledMap*. With the requirement FR.ENEMY_ATK, the enemy should follow the player, this is done by passing in the player into the update() method, then checking the x and y location then changing the direction of the enemy to move towards them, also when the enemy is close to the player, the enemy will then change into a random direction, to stop all the enemies from being in the exact same coordinate as the player. A **Enemy**.*changeSpawn*() method was created, so that we could spawn the enemy in a random location, this method generates a random coordinate, then calls the SafeMove method to check if the location would collide with the *tiledMap*. Within the **GameScreen** class, the update class will check when the last enemies were spawned and after some amount of time, it will call the **GameScreen**.*spawnEnemies*() method, which in turn calls the **Enemy**.*changeSpawn*() method. Similarly, the *update()* method also has another section of code for “ambushing” where a group of enemies can spawn around the player, also calling the *spawnEnemies*() method.

Other Changes

We also needed to change the overall **GameObject** class in order to achieve a headless program, which was necessary in order to make the tests work. The constructor for **GameObject** needed to be changed to remove the Array<Texture> and fps. The image handling was then moved into the existing *changeImage()* method inside **GameObject**. *changeImage()* (or *imageHandling()* in the case of Colleges), is needed to display any images of the objects onto the screen. Within Enemy, Player, College, creation of the Health Bar was also moved into a method, *createEnemyBar(), createPlayerHealth()*, *createCollegeBar()* respectively,also for testing purposes.

Just for a reduction in the GameScreen constructor, a couple methods: createPowerUps(), createWeather(), createColleges() were moved outside of the constructor then called inside the constructor, this was just for general cleanliness of the constructor, reducing the total number of lines inside the constructor.

Bibliography

[1] https://keiral11.github.io/Team12Website/media/a2Inheritance.png
[2] https://docs.google.com/document/d/16eiyZktg_zVBDlaH-orq-R1kQpcP0YBq7loDAX_8e2g/edit?usp=sharing
[3] Map Editor, *Tiled*, https://www.mapeditor.org/