Assessment 2:

Implementation

Mario

GROUP 28

Joseph Frankland

Anna Singleton

Saj Hoque

Leif Kemp

Shi (Lucy) Li

Hugo Kwok

3a

Assessment 1:
Git Repository: https://github.com/uoy-jb2501/ENG1-Pirate-Game/

.jar file: https://github.com/uoy-jb2501/ENG1-Pirate-Game/releases/tag/1.0.0

Assessment 2:
Git Repository: https://github.com/AnnabethS/ENG1-Phase2-Game

.jar file: https://github.com/AnnabethS/ENG1-Phase2-Game/releases/tag/latest

3b

For the implementation as part of our 2nd assessment, the 6 requirements we had to implement are:
Obstacles & Bad Weather, *Spending Plunder*, *Combat with other ships*, *5 powerups*, *Difficulty options* and *Saving/Loading games*

Obstacles:
The obstacles take the form of mines which are randomly distributed around the map, the amount of them is dependent on the game's difficulty. To achieve this we repurposed the CollegeManager class used in the original project, which already provided a solid basis for generation, which checks for tile types, which was repurposed to spawning on water tiles as opposed to the college's land tiles.
It also provided a check for distances between other objects, so it was repurposed for distance checks between both other obstacles and the player's college, this prevents the issue of the player spawning on a mine and taking damage/dying, as the mines spawn far enough away, the distance check for other mines also allows mines to be more spaced out and reduce the likelihood of there being large pockets of mines where navigation becomes impossible.

Bad weather:
Bad weather takes place in the form of a storm, which slows down all objects in the game by 20% and nullifies speed boosts, excluding cannonballs.
The behaviour is defined in the GameScreen class, the relevant variables are *isStorm, stormChanceInterval (default = 15), lastStormChance, minStormChance (default = 0.04f), maxStormChance (default = 0.5f), currentStormChance and stormTime (default = 90f)*
Each time logic() is called, it checks to see if the time elapsed is *stormChanceInterval* from the last time a check was called, so it happens per 15 seconds by default.
When it is time to call the check, it generates a random number between 0 and 1, if that number is less than *currentStormChance*, a storm is summoned and will dissipate after 90 seconds, assuming it isn't skipped.
If the randomly generated number is greater than *currentStormChance*, that chance is then doubled (capped at *maxStormChance*) and *lastStormChance* is set to the current gameTime, meaning the game will check again after another *stormChanceInterval.*

The storm can be skipped in 2 ways: By returning to the player's college and skipping it for free, or they can open up the Shop and spend plunder to skip the storm, which creates a toss-up between spending time by returning to the college, or plunder by buying the skip.

Ways to spend plunder:
During the game, the player will earn plunder by destroying AI ships and colleges.
The player can opt to open a ShopScreen by pressing N, here they are presented with several items to buy with plunder, in the order presented in the game:
Healing (repair to full health), Skip storm (skips the current storm, cannot be bought if there is no storm), **Damage buff (+20%), Reload buff (+20%), Speed buff (+20%), Max health buff (+25%), Health regen (regen 2HP per second)**, Timer extension (+30 seconds to timer)

Purchases highlighted in bold are one-off purchases, you can only buy them once per game and are permanent buffs for the remainder of the game.

To achieve this, the ShopScreen uses a Shop enum to determine which purchase is highlighted, when the item is bought, it will check that the player has enough plunder, if they do, it tells the GameScreen to apply the relevant purchase via the addPurchase() function, which then applies the relevant effect either to itself or to the player via their applyPurchase() function.

Combat with other ships:
Combat with other ships expands upon the EntityAIShip class, when it is either approaching or fleeing the player, it checks if the player is within the ship's *minDistance*, if it is, it calculates the direction it should fire based on the player's and its own position. Then spawns a cannonball and has it fire in that direction. It then has to wait *fireRate* seconds to fire another cannonball.

The main distinction here for firing cannonballs is that it does not utilise the EntityShip's fireCannon() function despite extending EntityShip, this is because that function constricts the direction of the cannonballs firing to just the left and right side of the ship, and would take considerably more time to rewrite in consideration of both the AI ships and the player ship, as well as require considerable adjustment to the firing behaviour, so to avoid unintended side effects, this firing behaviour has been relegated to just the EntityAIShip.

5 Powerups:
There are 5 powerups in the game, each with their own unique sprite and colour; SPEED, FIRING SPEED, DAMAGE, INVULNERABILITY, and RAM.

SPEED - 30% speed boost
FIRING SPEED - 2x firing speed
DAMAGE- 2x damage
INVULNERABILITY- Nothing can damage you
RAM - Pressing SHIFT will boost your ship forward, any objects that collide with the ship during a ram will be destroyed, without damaging the player, there is a speed check to ensure that the player's speed is low enough to initiate a ram, this prevents spam.

Powerup distribution is similar to that of obstacle generation, it is based off of the CollegeManager and employs similar checks between powerup objects, there is also some additional code to ensure that an equal amount of each powerup is distributed throughout the world. To determine the powerup's type, we use a PowerupType enum, which contains 3 additional functions;
powerupToString (turns the enum value into a string, which is used to display the powerup on screen)
fromInteger (takes an integer value and returns the enum value)
fetchTexture (returns the texture path based on the enum value, this allows the entity to texture itself)

To manage the state of powerups, the GameScreen uses a HashMap<PowerupType, Float> which represents the powerup and the time remaining. When a new powerup is collected, it

gets added to the list with the appropriate duration depending on the game's difficulty, the float then decays over time, and is removed when it reaches 0, which also removes the powerup's effect from the player. If a powerup with the same type already exists in the HashMap, the existing entry in the HashMap is reset back to the original duration.
The EntityShip is given several boolean flags to dictate which powerups need to be applied, and change the behaviour of the player accordingly.
When a new powerup is added, it uses the EntityShip's applyPowerup() function to allow it to apply the appropriate behaviour. The powerup behaviour mostly takes place in the form of booleans, and uses checks where applicable, such as determining the MaxSpeed and drag of the ship for speed boosts, or applying extra damage to cannonballs. The only case where this is different is the Reload Speed power up, which just changes the EntityShip's *reloadTime* variable directly, rather than using a boolean.
The EntityShip handles ramming by checking the speed of the ship, if it is below a certain threshold; the MaxSpeed of the ship, the maximum speed of the ship is greatly increased, and the drag is adjusted so that it doesn't immediately slow down, a large amount of velocity is then applied in the direction the ship is facing. While it is above another threshold, the ship is considered to be ramming, while in this state, if it collides with an entity, that entity is destroyed and if it is a damaging entity, the EntityShip will take no damage, the ship is also unable to carry out another ram while in this state, preventing spam and excessive acceleration of the ship.

Difficulty:
Difficulty is handled by using an enum, it has 5 options (EASY, NORMAL, HARD, GAMER, TEST) the key differences will be stated here:
EASY - Player health = **150**, Enemy health = **10**, Mine count = **100**, Powerup count = **100**
NORMAL - Player health = **100**, Enemy health = **25**, Mine count = **200**, Powerup count = **50**
HARD - Player health = **50**, Enemy health = **37.5**, Mine count = **300**, Powerup count = **25**
GAMER - Player health = **1**, Enemy health = **50**, Mine count = **500**, Powerup count = **10**
Powerup duration is doubled for GAMER difficulty, from 30 seconds to 60.
TEST - Player health = **100**, Enemy health = **25**, Mine count = **20**, Powerup count = **10**
This difficulty is used solely for testing to speed up initialisation of the game.
It is inaccessible in the game and has to be accessed via code.

The game's difficulty is selected at the DifficultyScreen, which is accessed when the player selects Play in the menu, then passed along to the GameScreen, which applies it to generation of obstacles and entities, the entities are also given the difficulty value, which allows them to adjust their health accordingly.

Saving & Loading games:
Implemented using libgdx's preferences system. This uses a simple key-value pair system in order to retrieve simple values from a file on disk. This is wasteful in terms of space, so would be inappropriate if the save game was larger, however due to minimising the amount of data needing to be saved, and instead regenerating lots of it at load time, this is a viable method. An option with lower storage cost, yet higher code complexity would be a binary dump, and reading that back in. Due to the small amount of data needing to be stored, preferences is a sufficient method for doing saving.

Additions and tweaks:

Reviewing the implementation feedback from Group 29's Assessment 1, one issue was the boat's controls being somewhat clunky, to remedy this and allow more flexibility in controlling the ship, an accelerate button was added, by pressing SHIFT, the EntityShip will now accelerate in the direction it is facing. This allows the player to continue moving without pressing any of the directional keys, and in any direction they wish.

The PirateGame class which handles transitions to new screens has been rewritten so that it only utilises a single function and an enum parameter, it requires only 1 instantiation of each screen, except GameScreen and ShopScreen as new ones of those 2 need to be created for each new game, this makes switching screens cleaner and reduces risks of memory overload by creating too many instances of screens.

Missing/Partially Complete requirements:
Some requirements that haven't been completed by Group 29 for their Assessment 1 will be listed below, and whether or not they have been completed fully, partially or left incomplete, with justification, others have been looked through and marked as such in the Requirements deliverable. All other requirements not listed here have been completed per the Requirements deliverable.

UR1(now UR_MOVEMENT): "*The player should be able to control a boat. Mouse and Keyboard controls must be available."*
This requirement is only considered to be partially complete as the game has no mouse controls whatsoever, everything is done entirely on a keyboard.
UR11(now UR_ACCESSIBILITY): *'The player should be able to identify the game without relying on colours'*
This requirement was not considered to have been fully met in Assessment 1, and this is for the most part the same for this assessment, as the graphical style has been maintained with the world.
Assessment 2's additions like mines and powerups could be considered for meeting this requirement, as they can be easily distinguished, mines are dark circular sprites in the brighter water, and each powerup has their own unique sprite, allowing players to easily distinguish between them regardless of colour.
SR2(now FR_ENEMY_BOAT_COMBAT): *'The system should have AI, complex enough to mimic a ship'*
As stated in Assessment 1, this appears to be a subjective requirement, we've been building upon the AI present in the project and adding shooting functionality on top of it, but no significant overhauls have been made to it as it's a good foundation to start with.
SR7(removed): *'The gameplay should be randomized each playthrough.'*
This requirement was removed from our Assessment 2 document, though it remains as part of the old requirements document from Assessment 1.
In Assessment 1 the game used hard-coded seeds to avoid the issues of creating games where colleges were inaccessible, we have kept this as creating an algorithm to search for valid seeds would be too time consuming and could potentially increase loading times considerably. The game could be considered to have more randomisation with the additional distributions of obstacles and powerups, so this requirement now seems more up to subjective opinion.