UNIVERSITY OF YORK
ENG1 – TEAM 18

Implementation

ROB ANDERSON
MORGAN ELKINS
ZIAD ISMAILI ALAOUI
SAM KNIGHT
SCOTT REDSHAW
GEORGE TONNERO

The code can be found at:
https://github.com/ENG1-GROUP18/boatgame

Primary modifications to existing code:

Due to the simple nature of the project we took over, there were very few changes that we needed to make to the existing code. When we did make changes, it was generally in order to resolve bugs and issues that were left over before we picked the project up ourselves.

The remaining updates to the code consisted of additions rather than modifications. The changes made are noted below along with the reason for the change.

id	Change made	Reason for change
1	Updated gradle dependencies and desktop launcher to use LWJGL 3 as the libgdx backend. The project previously used the older LWJGL 2 backend.	To make sure that we are using the most up-to-date version of libgdx for compatibility with online resources and documentation. Also fixes some issues associated with running the project on a linux based machine.
2	Modified the menu screen to make use of libgdx stage, table and label system when drawing text to the screen. This replaces the glyph system used before.	In order to meet requirements CR2 and USR11, the screen needed to be scalable to any size, something the old project did not do without resulting in misalignment of the contents of the screen. By using the table system, the screen could now be scaled and its contents would remain proportionally scaled and positioned no matter the resolution.
3	Implemented box2d hit detection to replace the existing system that used hard coded maths. (eg. pythagoras' theorem used for cannonballs).	The existing implementation used for hit detection between projectiles and the colleges was inconsistent, resulting in strange behaviour of the collisions. Often the projectile would travel directly over the college sprite without a hit being detected. This better conformed to the user requirement USR10 which required collision mechanics.
4	Changed the method of input for attacking from mouse to keyboard.	When demonstrating our project for the first part of the assessment we learned that trying to use a laptop touchpad for input can be inconvenient when trying to input precise input. We decided that changing the input to the arrow keys and movement to the wasd keys would be a good way of accommodating the mentioned use case.

Explanation of significant new features:

Entities

The main changes made to classes in the entities package were in relation to the hit detection. In the Player Class, the body attribute (Body) was added to facilitate work around collision detection without having to deal with overly complex mathematical solutions that are prone to fail. The hit box was mainly dependent on the shape of the sprite in previous implementations. This new direction allows us to be more flexible with the hit box, including resizing it, rotating it, etc… This addition caused a whole new set of operations to be added to cater to those changes, such as functions that determine the position of the player based on Body methods, or that determine whether the player (or any object inheriting a Body) was hit by another body of the same nature. This pompously smoothed the control of

intersections. Other additions were made to update the position of the Body as it moves along with the player.

That new feature was also adapted to other classes, namely, the College Class, the Bullet Class, the EnemyShip Class, and the SeaMonster Class. 9

Frameworks

The frameworks package contains classes whose purpose is to assist the entities through useful methods that are utilised by multiple entities, for example HealthBar and Hud. While we didn't modify the existing classes, we implemented a plunder system that contained simple getter and setter methods to manage the player's plunder.

Screens

The main changes to the screen package are the creation of a superclass BasicMenuScreen that all subsequent menu screens inherit from, a complete rework of the MenuScreen and extension of the PlayScreen.

By creating a BasicMenuScreen class that implemented Screen, we could ensure consistency in style across our menu screens through the use of extension. This class contained the foundations needed for the menu screens by defining a camera, viewport, style and stage that, once the class is extended, can be used to draw libgdx scene2d.ui label widgets to the screen with a universal appearance. This consistency would make it easier for the user to navigate the menus due to their similarity to those already encountered. It also allowed us to recreate multiple similar menu screens without repeating code, allowing easier modification that kept the screens uniform in style.

The BasicMenuScreen also makes use of a VfxManager from the post-processing library 'gdx-vfx'. The VfxManager is used to implement a range of shaders that help us to reinforce the retro style of the game, giving the appearance of playing the game on an old cathode ray tv. The library is used to intercept the output to screen of our game, apply a set of included shaders to it, then forward the result to the screen with the shaders applied. It allowed us to quickly and computationally efficiently achieve the desired effects without having to write the complex shaders ourselves. The shaders are configured in the BoatGame class, and added to a VfxManager object. When the BoatGame class is passed to each screen upon creation, those screens can make use of the VfxManager, allowing us to maintain a single instance of the VfxManager object and ensure a consistent visual style across the game that can be modified in a single file.

Secondly, the playScreen was extended to support the powerups. To do this we created an upgradePlayer function that consists of a switch case statement. This is called by the playScreen everytime a college is defeated and passes in a number indexing which powerup to do. The powerups in order are: unlocking the shop, improving armour by 25%, freezing enemies, giving the player 20 seconds of immunity and refilling health. To implement the shop unlocking, a variable is set by the update function to true. The effect of this is to validate an if statement in the update function which requires both this variable to be true and the M button to be pressed for the screen to be changed to the shop screen. The armour powerup is implemented by calling the player's scaleDamage function which scales the amount of damage they take when hit by a bullet by a parameter. Freeze is implemented

by calling the freeze function and then setting a time since freeze variable. This function calls all of the moving enemies' freeze function which in turn sets their current internal state to STAY which stops the enemy from moving (by setting its linear and angular velocity to 0) and shooting and then sets the isFrozen variable to true. Once a certain amount of time has passed, the playScreen then calls the unfreeze function and sets the isFrozen variable to false, which sets the internal state back to RETURN allowing the enemy to move and shoot again. The immunity is implemented by calling one of the player's methods - the variable and timer functionality for this are implemented within the player class. And the refilling health is done by calling the player's setHealth function and setting it to max health. In each case, on an upgrade an alert is written to the screen for a few seconds. This is implemented by a boolean, a variable holding the time since update and a number of calls to the Hud.

Thirdly, it was extended to support the save and load functionality. The constructor was changed so that now many variables are initialised to their analogue in the screen's GameState. Some of these variables are initialised in different ways depending on whether the screen is a new game or a load. For example, on a spawn the colleges are loaded onto the map by the addColleges function. This splits the map into 6 pieces and loads the colleges into a randomised location within its piece. But if it is a load, the college is simply loaded to its previous position. A getState function has also been added which creates a new GameState, updates its variables to store all of the information about the current game and then returns it. This is called whenever the game is saved and is implemented by calling the entities updateState function and assigning the variables not stored in the entities.

Fourthly, it was extended to support the shop and the macros purchased in the shop. As indicated previously, the shop is unlocked by killing the first college and is accessed by the button M, which saves the game and calls the changeScreen function. In the shop there are three macros for sale, i.e. you can pay to unlock buttons in the game that give the player an advantage. These are fire bullets that hurt ships, slime bullets that hurt monsters, and a button for trading plunder for health refill. These functionalities are handled in the handleMacros function. The fire bullets are toggled when both the variable hasBoughtRed is true and R is pressed. This scales the ship's damage by 2. Slime bullets and health refills are implemented in similar ways but with the buttons G and H.

Fifthly, it was extended to support bullet handling. Previously, the rendering and collision detection of bullets had been handled in the entities that issued the bullets. But this caused problems such as complicating the save and load process and making it difficult for bullets to persist after the death of their issuer. We thus moved the bullet handling into the playscreen in the handleBullets function. All bullets are now stored in a global array and updated based on their internal information.

Lastly, it was extended to support different difficulty modes. This was done by adding a setMode function that is called when the game is started. The parameter that is passed into this function is set in a screen before the game starts by the player, who has a choice between easy, normal and hard. To implement modes we call the method scaleDamage for enemy ships, colleges and the player; in easy mode we scale the amount of damage the player takes down and the colleges and ships up; in normal mode we leave it as it is; in hard we scale the player's damage up and the colleges and ships down.

Tools

The tools package initially contained a MapLoader used to load the tileset used for the background. One of the two additional classes that we implemented is B2dSteeringEntiity which contains the logic to provide intelligence to the enemy ships and monsters in order to adhere to the functional requirement FR17.

The other class implemented is WorldContactListener, which is used to assist with the hit detection required to meet the user requirement USR10.

GameState

GameState is a new class created to save and load the game. It uses the google JSON library GSON to serialise java objects into json. The advantage of this was that GSON can serialise java objects of any kind. However, when we first used it it had difficulty parsing both constructor functions and functions in general, and ended up serialising the wrong information. This led us to use a C-like style for the GameState; it has no constructor function and no getter or setter functions; instead it uses public variables for getting and setting, and so resembles a struct. Within the class, these variables are split by comments into 6 sections: info for player, ship, monster, college, bullet and test.

Boat Game

The class BoatGame initially extended Game but its only purpose was to set the initial screen using `setScreen(new MainMenuScreen(this));`. To accommodate the additional screens that we were to add, we implemented a changeScreen method that is passed an enum screenType. Since BoatGame extends Game, we can make use of the setScreen method to orchestrate the game screens. If trying to change to a screen type that had not been instantiated, the screen was simply created then changed to. We also added loadGame and saveGame methods. saveGame creates a new Gson object and calls the playScreen's getState function. This returns a GameState object which is then jsonified by the Gson object and saved to the preferences file that comes with LibGdx files. This is indexed by the single parameter passed into saveGame. The loadGame method creates a Gson object and gets a string from preferences indexed by a parameter passed in. If there is no such string it creates a new playScreen with a standard GameState; otherwise it turns the string into a GameState class which it passes into a new playScreen instance.

Features not fully implemented:

The game meets all of the feature requirements outlined in the product brief.

The requirements that haven't fully been implemented are as follows:

- USR9 "There should be an option for controls that the user wants to use". The control scheme accommodates the full range of perceived use cases. Therefore providing the option for an alternative control scheme, while missing, has very little to no impact on the functionality of the final product.
- FR13 "There should be sound effects". The game doesn't use sound effects however this was assessed to be a low priority requirement and as such has minimal impact on the games functionality.