Sea of Geese

Architecture

Concrete Architecture

To represent the implemented architecture of the game, we have used the intelliJ IDE to generate two UML diagrams. We have represented the architecture in this way in order to allow easy comparison to our UML diagram we created for Assessment 1 [1] at the bottom of which is also a justification. The first diagram shows the classes in the main package, including inheritance [2]. The second diagram shows the relations between classes [3].

We have omitted MainGame from [3] in order to de-clutter the diagram. This is because all classes and subclasses except StaticInteractiveCollisionObject, IslandEdge, and WorldCollisionCreator relate to MainGame, either directly or indirectly.

Justification

The main parent class of all the game is MainGame. MainGame is the first class that is called when the game is run and contains most of the core functions that are required. Examples of this include the screen switching and the enemy initialisation. As a result, all classes except StaticInteractiveCollisionObject, IslandEdge and WorldCollisionCreator are children of this class. We were required to do this as we had to access functions such as changeScreen and the Player methods from anywhere in the game. In our original class diagram [1] we didn't have a specific main class, which was an oversight. By having a main class, we can ensure that the different parts of the game can access most other parts of the game that are required in order to work correctly.

The next largest class[3] is Character. This is used as the parent class for all of the "interactable" units within the game, such as the player, enemies and colleges and departments. Similarly to our abstract architecture [1], the Character class contains the base variables such as x and y position as well as methods that are used by all of the character classes, such as getStructureHealth() and getxPos() [2]. The classes Player, Ship (which represents the enemy boats) and Buildings (representing the colleges and departments) extend this class in order to have the base attributes as well as more specialised variables and methods.

GameContactListener essentially takes the role of the encounteredPlayer() method from the NonPCharacter class in the abstract architecture [1], by implementing ContactListener and when to objects collide, creates fixtures, checks their object type and then does something based on the type received. If the player collides with a building or enemy ship, the class CombatScreen is called.

CombatScreen carries out the combat within the game. It is called by either Ship or Building when its collided with. Our abstract architecture [1] had a combat class and roughly does the same thing, except it doesn't handle the graphics side of it, which our concrete version does. It takes a character and an instance of the main game and allows the player to attack the enemy character, dealing damage. Every time the player attacks, the enemy also randomly does one of two attacks.

Hud is the class which, when called, creates a new viewport and a table and writes to it the current Gold and Point values of the player instance, as well as updating this whenever these values increase. This allows the user to see their points and how much gold they have at any given time, whilst sailing around the map.

The MainScreen class is the class which contains the actual "playable" game. Again, we didn't have a version of this in our abstract architecture [1]. By using functions such as handleInput(), this class monitors the user's input methods and updates the x and y position of the player, as well as rendering the map. This also uses the Hud class to create a display for users which shows important information [2].

Quest and QueststateController handle the planned quest system. While the code of Quest is complete and has been tested for this aspect of the game, QueststateController is not currently finished - this being a task for later assessments. Quest allows for the definition of individual quests - e.g. what the quest asks the player to do, whether the quest is repeatable or not, and what the reward is to the player for completion of the quest. QueststateController handles the player's interactions with instances of Quest. This is done by reading from a file which contains the details of the quests. Quests are either active (can currently be completed by player), completed, or "unbegun" (cannot currently be completed by player until certain conditions, such as completing another quest, are fulfilled). Neither class has a parallel in [1] as it was a design decision made after the end of Assessment 1.

StaticInteractiveCollisionObject is used to define the collision boundaries of the boundary of the map, as well as of the islands in map.

For all of our "screen" classes, we didn't have any representation of them within the initial abstract architecture [1] as we hadn't thought of how we would deal with the different screens that would occur when playing the game.
The LoadingScreen class is called right at the beginning of MainGame's create function, so it is the first thing that the user will see. It creates a new stage and displays a splash screen of our team's logo for 5 seconds then calls MainGame's changeScreen() method to switch to the main menu.

WinScreen is called when a specific event occurs, presenting the user with a "Win Screen", to let them know that they have completed the game. When the "Halifax" building is defeated, CombatScreen calls MainGame's changeScreen() method to switch the screen to the WinScreen class screen. Control is only given to the player in the form of the escape key, which is used to quit.

EndScreen is very similar to WinScreen in the way that it requires a specific event, however this one is seen when the player is in combat and their health reaches zero. Again, CombatScreen calls its parent's changeScreen() method to switch to EndScreen and again, the user has to press escape to quit.

MenuSceen is called by LoadingScreen after the splash screen has been displayed for 5 seconds and contains a table that the user can click two buttons in: New Game and Exit. New Game will call the changeScreen() method to change to the MainScreen and allow the game screen itself to be rendered, allowing the user to play the actual game.

Bibliography

[1] Assessment 1 UML Diagram (Figure 2).
https://davidjnorman.github.io/SEPR/assessment_1/Arch1.pdf
[2] UML Classes Diagram.
https://davidjnorman.github.io/SEPR/assessment_2/uml_classes.png
[3] UML Relations Diagram.
https://davidjnorman.github.io/SEPR/assessment_2/uml_relations.png