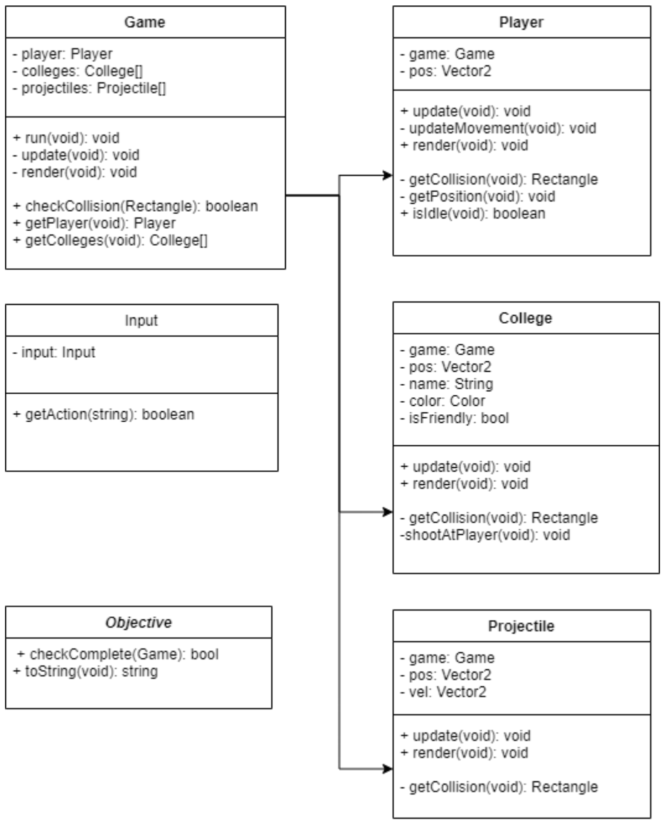
3. Architecture

Team 10 | YorKorsairs

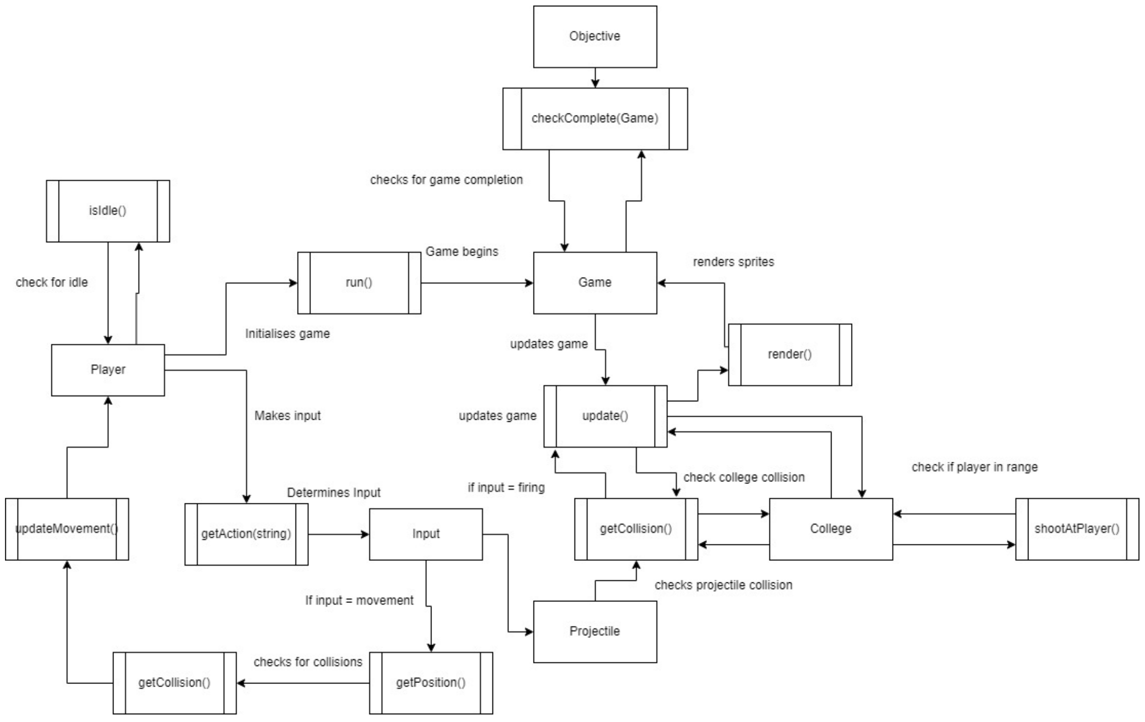
Abdullah
Tom Burniston
Omer Gelli
Adam Kirby-Stewart
Sam Mabbott
Benjamin Stevenson

A. Representation

The initial abstract UML class diagram of our game:



The abstract data-flow diagram for our game:



Our initial abstract representation contains 6 main classes, these are the classes we have deemed as core components which we will need to meet the objectives. We have populated these classes with basic methods which we believe each class will need.

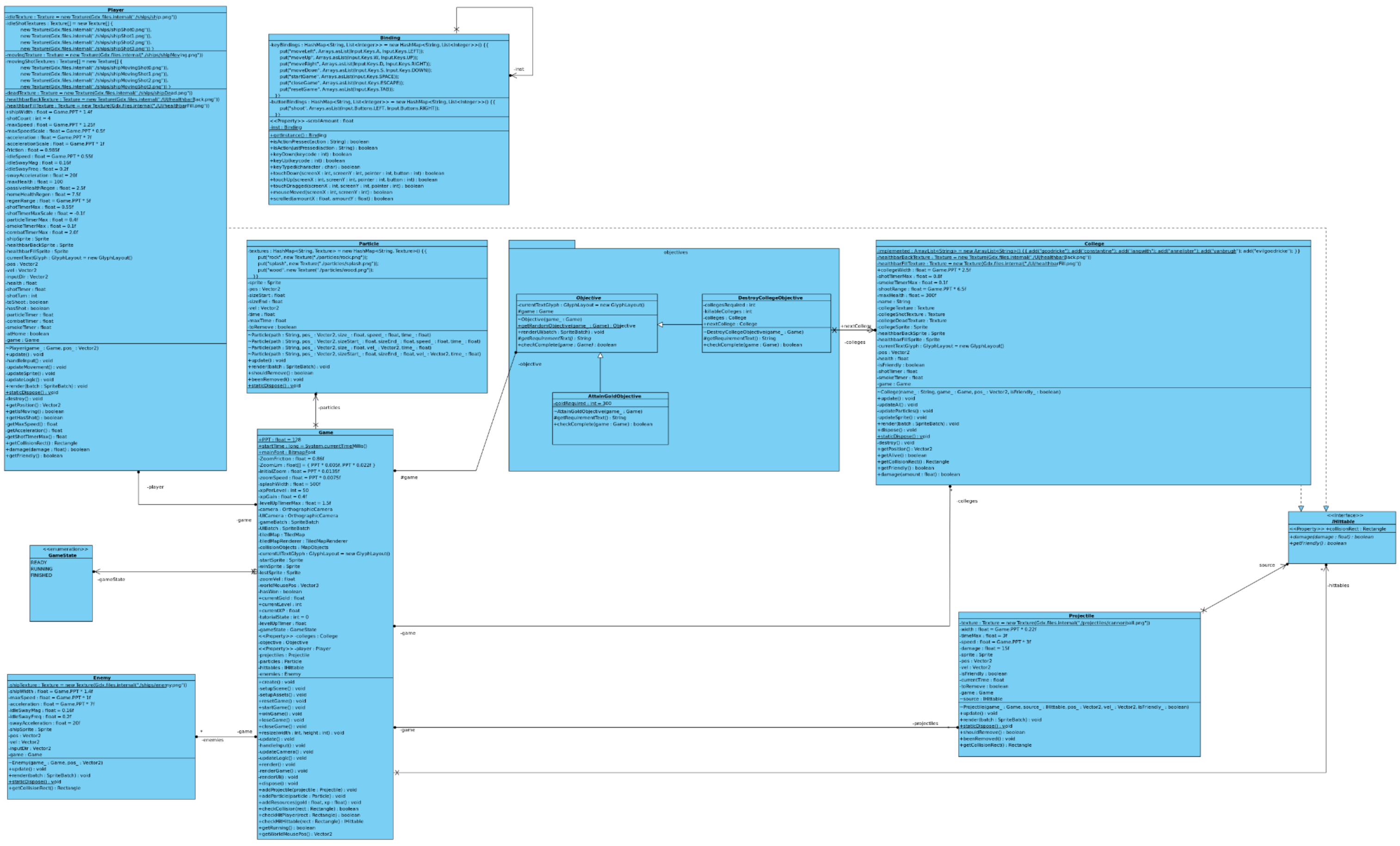
The 6 basic classes are:

- Game - The game class will be the main class for the game. It will keep references to a single Player class instance, and the multiple College instances. Within the Update and Render functions, it will call each instance's update and render functions. During construction, the Game class will load the tilemap, as well as extract the collision objects from it.
- Player - The player class will handle the players ship within the game. The player class will keep a reference to the Game, to allow it to access it's public functions. One of the main functions that it will access, is the checkCollision() function. This allows the player to check whether it is currently overlapping with the terrain, and react accordingly.
- College - The college class will handle all the colleges within the game. The class will keep reference to the game and will use its public functions. Like the player class it will make use of the render and update functions as well as have a shotAtPlayer() function to allow for combat.
- Objective - The objective class will keep track and check for completion of the game and individual objectives. This class will have functions to check the game state, and text which indicates the requirements.
- Projectile - The projectile class will create and manage all projectiles within the game. Projectile will also have its own Update() and Render() functions, which will make sure it continues on moving in the direction it is created with. It will also contain a reference to Game, which allows it to use the checkCollision() to be able to detect when it hits a Player or a Wall.
- Input - The input class will handle the keyboard and mouse inputs from the player and, through mapping, ensure the correct action is taken within the game. This will include moving the player, firing projectiles and resetting the game.

The data flow diagram above also gives a basic overview of how the classes interact with each other using the functions mentioned.

We created our abstract class diagram, and abstract Data-flow diagrams on diagrams.net (formerly draw.io) flowchart maker. The concrete class diagram was created using Visual Paradigm. Both the concrete and abstract class diagrams use UML as it was deemed appropriate for this project because of its tools which it provides for representing object oriented design.

This is our concrete UML class diagram:



A higher resolution version of this diagram is available on our team's website. (https://aj141299.github.io/YorKorsairs/assets/images/Concrete%20Class%20Diagram.png)

B. Justification

The game takes place in a single screen, and as such a system for swapping between and handling multiple screen states for menu / game etc. will not be necessary; the game can be handled within a single Game() class. Variables are encapsulated by being made private, and providing public getter / setter functions when necessary. This means that they cannot be changed or accessed without the class's knowledge, allowing for maintenance of state.

The first major feature we added to the game was the implementation of health for botht the colleges and the player. This was done by adding private attributes to the Player and College class. These attributes are Health and maxHealth. They will be used to give the player visual feedback on the health of the ship and enemies as well as passed through the game class to check for changes to the Objectives or GameState such as the health of a college or the player reaching 0 (F009 and F010). The updated GameState or objectives will be shown to the player via updated visuals or splash screens (F011).

We also added health regeneration for the player to the game. This is done through the Health attribute of the player as well as the isFriendly method and a proximity check to the friendly college. The player's health will regenerate passively while in the open ocean but will regenerate significantly faster while ‘docked’ at a friendly college (F018 and F019). XP is also accumulated passively as the player sails around the map via the Game classes xpGain method. (F002).

The 2nd major change that was made when we began developing the game was the implementation of the checkCollidedHittable() function. This is a function that checks a Rectangle against everything that can be hit and damaged. This would also mean that there would need to be an IHittable interface, which Player and College can implement that would check the difference between player friendly and enemy objects (F007).

Another change that was made was the implementation of the IHittable class. The player (and college) will implement this IHittable class and in turn, implement the hit(float damage) function. This will deduct the damage from health, and also detect when it has run out of health.

The benefit of an interface over using a default implementation of the function is that the player and college can react differently - the Player will update the Game class when it is destroyed and the game state will change, meanwhile the College will destroy when it runs out of health. The visual update of destruction and health bars can be handled in other functions either in render() or update(). The College will need to update the Game class however, to allow for keeping track of the objectives such as the college becoming captured. This should also provide the player with points as well as gold.

Next the UpdateAI() function for the colleges was expanded to include the primary logic for detecting the distance to the player as well as the direction to the player. The function will then decide whether to shoot at it or not. This uses the Game classes getPlayer() to acquire the reference, and then performs simple conditional statements and maths to figure out what to do. When deciding to shoot, this will in turn create a Projectile and call addProjectile() on the game class.

Then we adapted and expanded upon the Objective class. Given that the primary purpose of the game is to achieve a specific and different objective. We decided to implement Objective as an abstract class. Other Objective classes will inherit from this abstract class. This includes classes such as DestroyCollegeObjective, KillShipsObjective, AttainGoldObjective etc. The Game class will need to retain a reference to an Objective class. This meets requirements F015 and F017 by keeping track of the players progress. By having objectives that are multi-step and are tracked throughout it increases the game time to help achieve NF002 while simultaneously not drastically increasing the difficulty level (NF003).

We then added an Enemy class to add enemy ships to the game. This is to meet one of the requirements set in the design brief. In the current version of the game this class has limited functionality and only creates an enemy ship that can move and check for collisions with the map border. At present these ships do not have the ability to engage in combat and cannot be destroyed. The class has been built in a way that allows for its expansion in these areas.

We also added a Particles class which creates particles within the game, it has its own render and update functions to do this.(F014) This was done to make the game more appealing visually and to attract the target audience. (NF001)

We also added on screen advice and stats for the player so they can easily tell how much gold, health and XP they have along with any tutorial hints such as game controls or mission objectives.. This is done via the tutorial state and UIBatch in the Game classand will call the current objective progress as well as Player attributes to display the current data on the UI. (NF017 and NF018).

Furthermore, we changed the Input class of the abstract diagram to a new class called Bindings, this handles all keystrokes from the player and has the functionality to change the keys for different actions within the game. The ability to change keybindings without editing the source code could be added here but given it is not a requirement we did not add this feature. This class allows us to meet F001,F003,F004, F012, F016, F018 and NF007 as the class allows for multiple inputs as well as keymapping. This ensures the game acts in the correct way and is customisable to the players needs.

We also added combat to the game using the aforementioned Hittable class as well as an is friendly boolean attribute to check if a nearby college is friendly. The college also makes use of the same attribute to distinguish between enemy and friendly ships. The college can also check if the player is within a certain distance and will call the projectile class to fire at the player when appropriate (F008) When a college is destroyed the player is given gold and XP. (F005)

The game has also been designed to be completed single player (NF004) and does not have the capability to access any kind of network (NF011, NF012) this also stops any malicious attempt to alter the game via a network.The game has also been designed to be colourful and bright which helps attract our target audience (NF001). This also allows for the game to be user friendly and easy to navigate. (NF005). The colleges, enemy ships, loot and map have distinct colours and shapes, there’s also no overlap between red and green usage throughout the game, this allows colour blind people to play with less difficulty (NF006). The game also does not include any flashing images or lights to help reduce the risk of causing a seizure while playing (NF009).

References