3. Implementation

Team 10 | YorKorsairs

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

Additional Implementation

The initial overview of the code and documentation brought to light a range of requirements / features required for *assessment 1* which were not met adequately, or some not at all. A subset of these did not require extensive architectural or codebase changes and the vast majority of the systems for each were in place and were missing basic functionality / implementation::

- The **Ship** component had functionality for gaining ammo / plunder, but this was not fully implemented. To meet **UR_EARN_MONEY** and to aid **UR_SHIP_COMBAT** and other combat based requirements, renewability for ammo and gaining of plunder was implemented from quests and destroying colleges by correcting the **addAmmo(), addPlunder(), reload(), and other** functions

- The above implementation required that the **Cannonball** entity worked correctly - This required fixes such as altering the system for removal of cannonballs, as well as adding a timer for removal. This ensured that **FR_BULLET_TRAVEL** was correctly implemented and also allowed **UR_HOSTILE_COLLEGE_TAKEOVER** to be sufficiently implemented.

- Each of the individual **-Manager** classes followed a system of static initialization on request, which meant that simply replacing the **Game, Menu,** and **End** screen didn't reset the state fully. A reset() was added to each to implement **FR_GAME_RESET.**

- The initial menu screen UI was too small, so to help fully ensure **UR_GAME_OBSERVABILITY** the menu UI was made larger and more clear. In addition to this, ingame the UI was attempted to be made scalable altho given technical difficulties and a lack of full requirements, this was no fully implemented.

- The missing requirements of **FR_XP_TRACKING** and **FR_XP_UPDATE** were both implemented, and following the improvements outlined above when allowing for gaining of ammo and plunder on the **Pirate** class, gaining XP from destroying buildings was implemented. **UR_GAME_OBSERVABILITY** was also perpetuated and UI for seeing your XP was added.

- When implementing the above UI additions, quest tracking was implemented (for requirement **FR_BOSS_UNLOCK_TRACKING**) through the means of a new UI label. While the original intent of this requirement was a boss, this was decidedly removed from the game, however the requirement was kept as tracking quests was deemed important. This addition was essentially trivial given the systems for tracking quests (in the **QuestManager**) and the structural code for displaying text on the screen was already set up.

- The final of the important missing requirements for assessment 1 - destruction of colleges - could now be implemented. Using the plunder / ammo / xp changes for the Pirate class as well as the implementation of a new **HealthBar** component and entity, requirement **UR_HOSTILE_COLLEGE_TAKEOVER** could be met.

Once the unfinished / missing requirements for assessment 1 above were implemented and corrected, our focus changed to the additional requirements for assessment 2. These often required full new systems (Such as the **EventManager** and the new **ShopScreen**), however there were no major changes, rather just extensions to current systems.

- Previously, there already were other ships as well as a state manager / pathfinding system that handled the AI. Fortunately, the **NPCSShip** entity already had a **Pirate** component due to deriving **Ship**, meaning simply calling the inherited **Shoot()** function allowed these ships to engage in combat, to satisfy the requirement **UR_SHIP_COMBAT**. Furthermore, due to the abstract implementation of these components, the **HealthBar** implemented for colleges could be extended to all ships meaning both the player and enemies could have variable health. This did require the addition of damage to be passed into the **Cannonball** class through the **fire()** function.

- The tasks of implementing the requirements **UR_OBSTACLE_ENCOUNTER, UR_WEATHER_ENCOUNTER**, and **UR_POWER_UP** were able to overlap through the use of a general purpose **EventManager** class. This acts similarly to the other **-Manager** classes in that it is initialised during the relevant **-Screen**, and also has functionality for **reset()** to ensure **FR_GAME_RESET.** This class's primary purpose is to randomly spawn classes deriving from the **Event** class (Such as **Storm, Monster, Powerup**), at random positions dictated by the settings file. These would be ensured not to overlap and follow configurable durations and cooldowns.

    - One of the primary requirements of assessment 2 was **UR_WEATHER_ENCOUNTER,** which has been implemented as **Storm**, a class which is derived from **Event** and is handled by the **EventManager**. Due to architectural decisions this class does not need to handle much functionality and simply detects and handles overlapped ships.

    - Similarly for the **Monster** class (as the implementation of **UR_OBSTACLE_ENCOUNTER**), this derives **Event** and is handled by, and instantiated by, **EventManager**. This contains slightly more complex code than the **Storm**, in that it directly interfaces with the **Ship** it follows, as well as updates its **Sprite** to show animation, on its' **Renderable**.
    - A **PowerUp** class was also added (to implement **UR_POWER_UP**) this reives Event and is handled by and instantiated by the **EventManager.** This code randomly generates one of several powerups such as an increase in health or a level up. These changes are applied directly to the player ship and can mostly be seen via the in game information.

A shop screen was added to fufill the requirement **UR_SPEND_MONEY** this extends the Page class similarly to the main and end screens. The shop screen is accessed and closed via the 'E' key and contains buttons that remove gold from the player when clicked. This action also gives the player a boost in a stat or alternatively more ammunition depending on the button clicked. The actions have built in checking to stop the player going into the

negative gold-wise as well as a check to stop the player spending on a heal when already on full health. When the shop screen is activated the main game and all entities within it are suspended. This makes the game effectively paused as the **Gamescreen** is frozen in that frame temporarily.

A difficulty selection button has been added to the main menu so that the games difficulty can be changed (**UR_DIFFICULTY**) before the game begins. This required a new variable in GameManager as well as checks throughout the code. These checks determine the difficulty level and in turn the damage done by enemies such as **Monster** as well as the rewards given by the **PowerUps**.

Significant Changes

The main significant change made to the codebase was the addition of **removeComponent()** and **removeEntity()** to the **EntityManager**, as well as changes to the data structures within **RenderingManager**

Previously, it seemed that they intended to never have to delete components and entities, as shown by the cache of cannonballs in the **GameManager**. While this approach was reasonable, use of caches was cumbersome and was not deemed reasonable when attempting to implement the **Events in** the **EventManager**; it is required that the events were deleted / instantiated constantly and of different types, and would have required large caches of each potential variety.

To allow for the removal of components and entities, not much change was required in the **EntityManager**. However, the **RenderingManager** was reliant on indexes within the previous **layers** variable to keep track - this was not reasonable with an array whose elements may be removed. Instead, the **renderItems** and **layers** variables were combined into **renderLayers**, whose elements were lists of Components, each representing an individual **RenderLayer**. This meant that rather than referencing an index the layers directly held a reference to the component, and cut out unnecessary potential issues. The functionality of these variables were essentially retained, and to external classes it was unchanged, apart from the addition of the **removeItem()** function.

Because of complications of overlapping managers, the **Renderable** class required additional checks for removal. This meant adding **remove()** functions to both Entities and Components, so the **Renderable** component could detect being removed and propagate this to the **RenderingManager** accordingly.

Unimplemented

A subset of the key requirements for assessment 2 were not implemented - **UR_SAVE_STATE** and **FR_CROSS_PLATFORM_GNU_LINUX**. These requirements were intended to be implemented however each imposed challenges that were not met during this development period.

UR_SAVE_STATE posed a significant challenge to the complex architecture of this inherited project. The large quantity of different **Entity**, **Component**, and **-Manager** classes all managed by a separated top level consisting of **-Screen** classes as well as the **PirateGame** primary class lead to difficulty understanding the best way to serialise the content of the game, and seperate the key information out into a file. If this information was accessible, the file writing most likely would have been handled by a JSON, either using in-built functionality in the libraries or adding additional 3rd party libraries.

FR_CROSS_PLATFORM_GNU_LINUX instead exposed different difficulties with the project - the primary structure of the .gradle files, and the separate **DesktopLauncher** meant that it was complex being able to understand where was best to separate the code. Furthermore, the **DesktopLauncher** class already implemented seemingly hacky fixes to allow it to work on mac, so therefore conscious choices during management and planning of development in our group were made to leave this functionality until later on in the development. Furthermore, this requirement did not seem as high a priority as the others, given workarounds such as running it on mac / windows were available.

Both of the above outlined requirements have a list of separate issues, but both also succumb to the limited time frame that the project was set to. Due to each of their agreed upon low priorities they were left unimplemented.