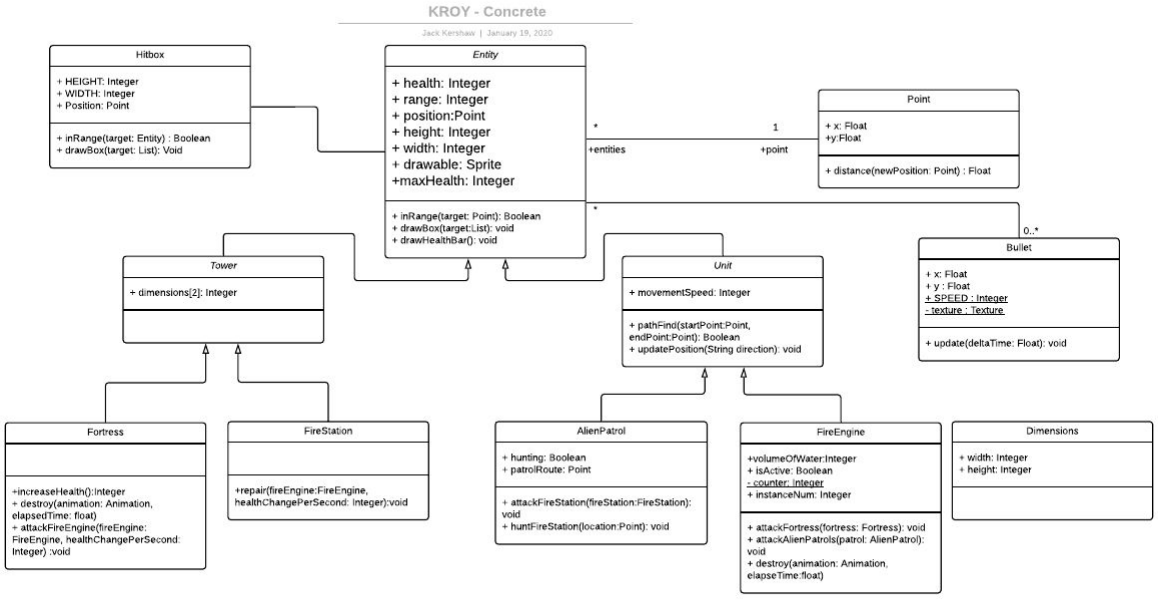
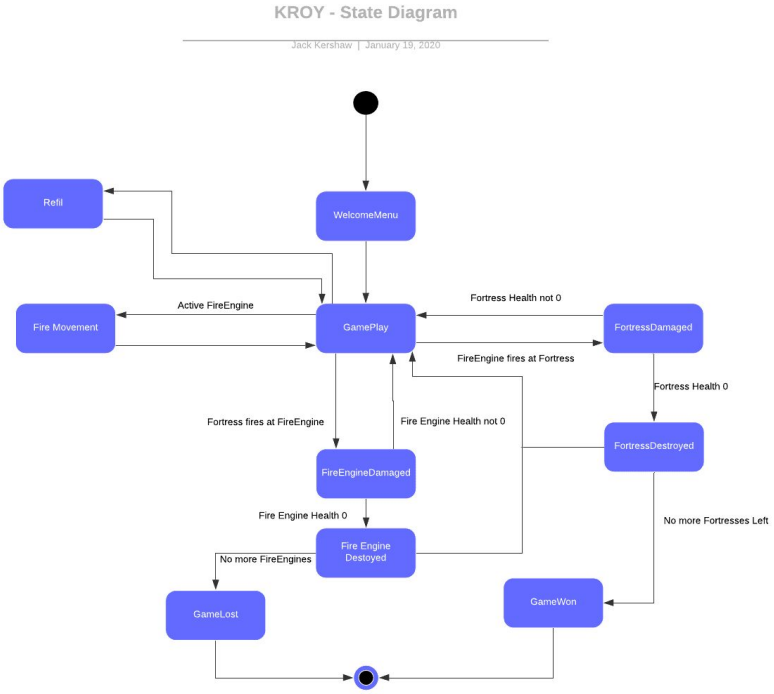
Module	SEPR
Year	2019/20
Assessment	2
Team	Dalai Java
Members	Jack Kershaw, Max Lloyd, James Hau, Yuqing Gong, William Marr, Peter Clark.
Deliverable	Architecture

Architecture

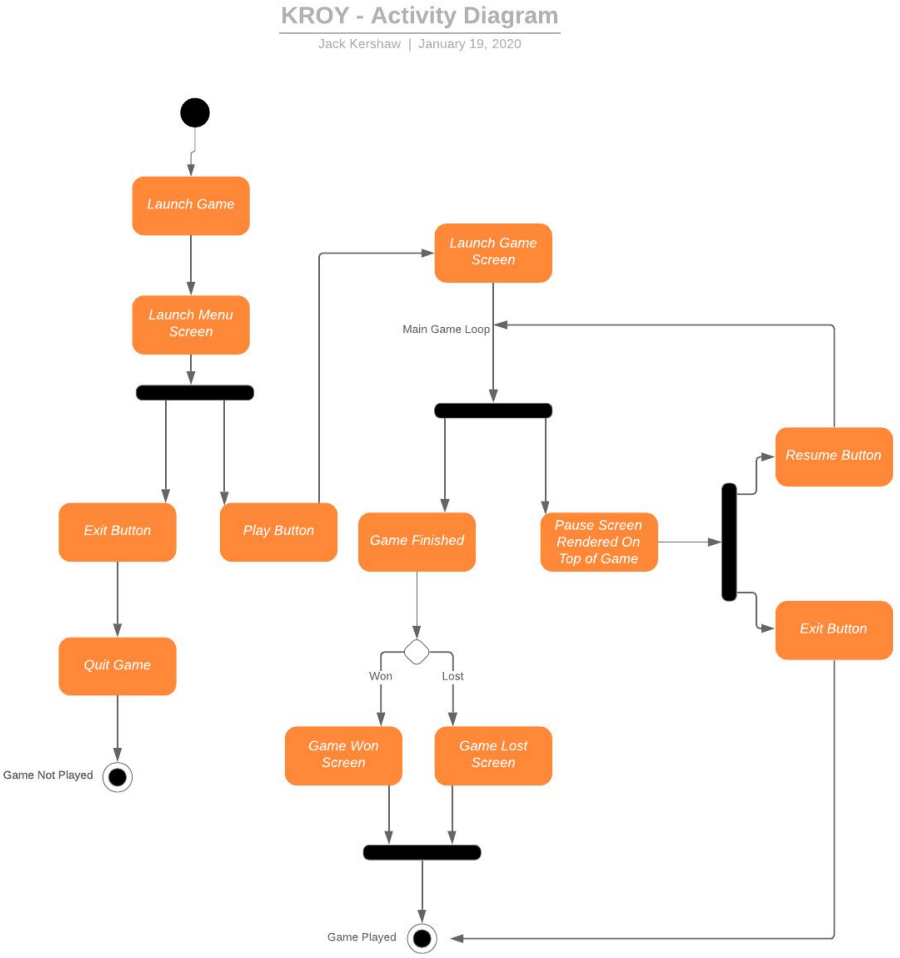
To show the relationships and structure between the different classes within our game we produced a concrete code architecture using UML in a collaborative tool called LucidChart. The main reason we chose to use lucid chart over the text based plant UML was the ability for multiple people to be able to easily view and edit the architecture in real time. The game has been developed in Java using the LibGDX library. This provided the functionality for the graphics, loading of the tiled map and a variety of other features. The main inheritance structure from our abstract architecture has remained but several of those classes have had additional methods and attributes added. The classes created for the library all have dependencies on the “base classes” and use these to implement the underlying functionality of the game. By changing the state of these objects the main game loop is able to update the screen to display the current state of the game.



The state diagram has also been slightly reduced in complexity as at the current time the alien patrols have not been implemented as well as the minigame. The rest of the game works almost exactly as shown in the original diagram as our teams concept for the game remained unchanged.



For this assessment we also produced an activity diagram in order to show the flow between the different game screens that we have been using to display different parts of our game.



Justification

The structure of our program has quite closely followed the UML class diagram produced as part of the abstract architecture produced. The majority of the methods needed were mentioned in the abstract architecture but some were more complex than expected and so had to be broken down into multiple smaller parts. Some key classes were entirely missing from the abstract architecture such as the projectiles of both the fortresses and the fire engines. These are key to the game as this is what causes the Entities to take damage and make up several functional requirements(FR_ENGINE_DESTROYED, FR_FORTRESS_DESTROYED), as well as without them the game would not be able to be won or loss(FR_GAME_WIN, FR_GAME_LOSE).

The class Bullet was added to our architecture to account for this. The class is responsible for both rendering the bullets as well as calculating where the bullet should appear next. This is based on the deltaTime between frames and the SPEED constant. This ensures that if there was to be a frame rate drop the bullets speed will remain constant. The fact that the Bullets are an independent class to the entities allow for the damage to be dependent on what fired it. This is also responsible for deciding whether the bullet hit the enemy fortress.

Parts of the abstract architecture were also entirely emitted due to them not being required for this deliverable. This includes the Minigame, to reflect this change a modified state diagram was produced in order to demonstrate the effects that this change

Another class, Hitbox was also required for the game as this has been used to register ranges, hits and prevent the Fire Engines from being able to drive off the roads. The class is also able to render the hit boxes on the screen which was invaluable during testing and debugging of the game. This ability is able to easily be enabled or disbaled with the use of a single variable in GameScreen called testMode, a simple if statement checks the value of this and if true runs the drawBox method on each hitbox used for limiting the movement of the fire engines as well as those of the fire engines and fortresses. This class did not require any relationships with other classes.

The FireEngine class covers almost all of the functional requirements presented to implement UR_FIRE_ENGINES. FR_NUM_OF_ENGINES has not been fully met due to the requirements of this initial phase of development. The functionality in order to meet this though does exist and so can quickly and easily be amended when it is later required. In order to allow this a static counter attribute has been included which is only initiated once and so can be incremented on each instance of the class that is created allowing a unique reference to be assigned to each fire engine. Although not required for the game to correctly function, during testing it makes it possible to discern between each fire engine from a console output. On creation of each fire engine speed, water capacity, range and health are required which allow each fire engine to have a unique spec as stated by FR_ENGINE_SPEC_SPEED, FR_ENGINE_SPEC_WATER, FR_ENGINE_SPEC_RANGE and

FR_ENGINE_SPEC_DAMAGE. FR_ENGINE_SPEC_DELIVERY_RATE is directly tied to the speed of the fire engine and so is also unique for each instance.

Fortresses have been implemented in according to the requirements established in the previous assessment. As required by the briefe we have included three fortresses, not matching FR_NUM_OF_FORTRESSES. The fire engines can only be shot while in the range of the fortresses(FR_FORTRESS_IN_RANGE) with the other requirements all being fully met as well(FR_FORTRESS_SPEC_WATER etc.).

The activity diagram shows the way in which the libGDX library has been used within the development of our game. On launch the MainMenuScreen is loaded which only is used to either exit the game without playing or to start the game. On clicking of the play game button the resources in the MainMenuScreen are disposed of and the GameScreen is launched. This class also acts as the input processor responsible for detecting inputs within the game such as mouse clicks and use of the keyboard. On initial loading, the required textures are loaded, and attributes are initialised with many constants to do with the position and dimensions of fortresses as well as the pauseScreen. The main game loop is contained within the class, the method render() will act as this within libGDX. On each frame this method is called, initially it checks for which fire engine is currently active before checking to see if SPACE has been pressed. If it has bullets will be fired based on the location of the mouse relative to the fire engine. Movement and rendering of the map is also dealt with in the class. It also deals with checking which fortresses or fire engines have no health left. On each frame it checks each of these and any that have been destroyed will be removed from the game. Once the game has been completed it loads the GameOverScreen and either displayed Game Won(FR_GAME_WIN) or Game Lost(FR_GAME_LOSE) depending on the outcome of the game(FR_NOTICE_GAME_OVER). This main method is also responsible for cleaning up any objects that are no longer required such as bullets that have traveled beyond their range(FR_ENGINE_SPEC_RANGE) as well as destroyed fire engines and fortresses.