Architecture

Mozzarella Bytes | Team 18

Assessment N°1

Daniel Benison

Elizabeth Hodges

Kathryn Dale

Ravinder Dosanjh

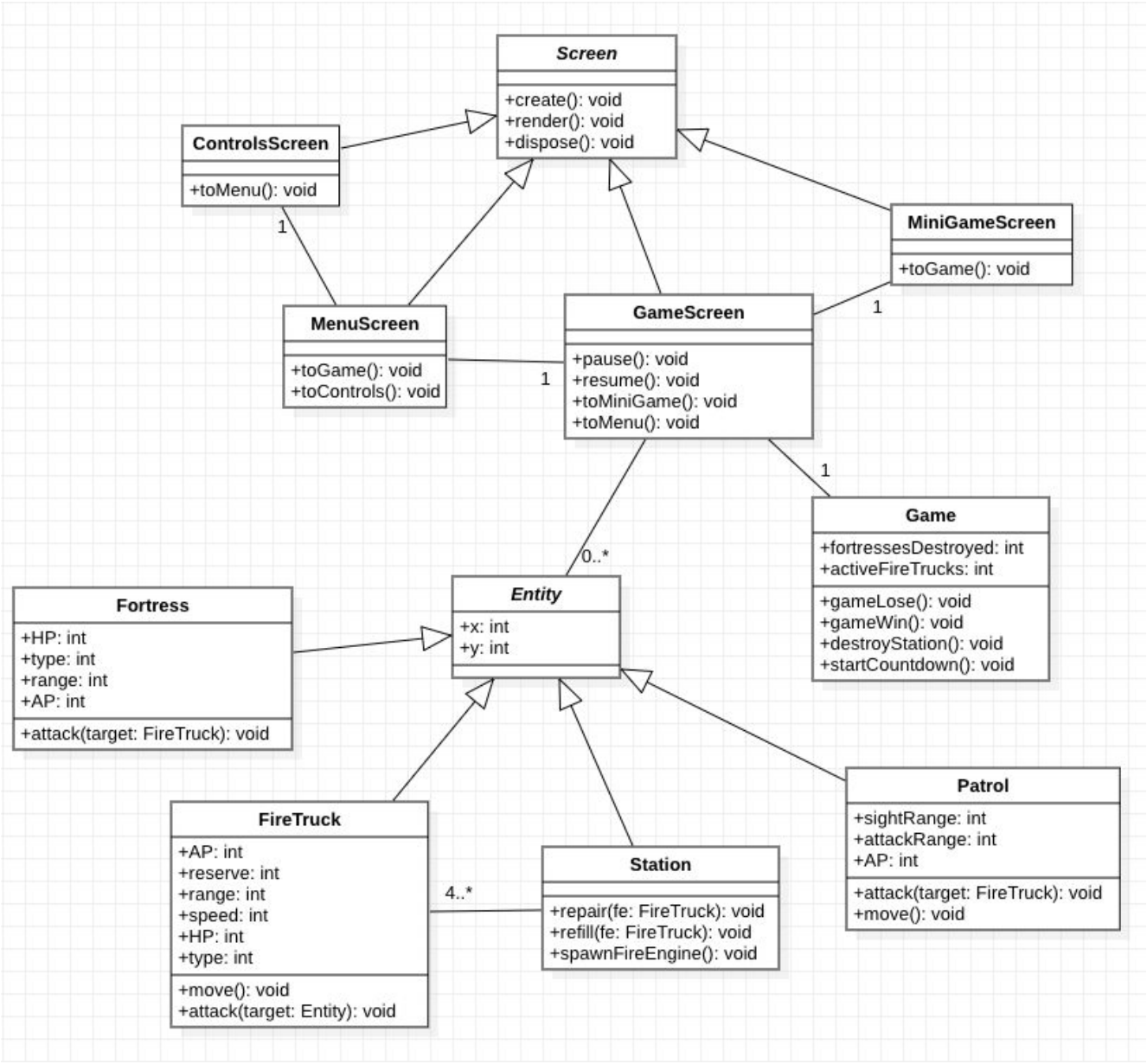
Callum Marsden

Emilien Bevierre

Architecture

We used the software StarUML to create the following UML class and state diagrams.

Class Diagram

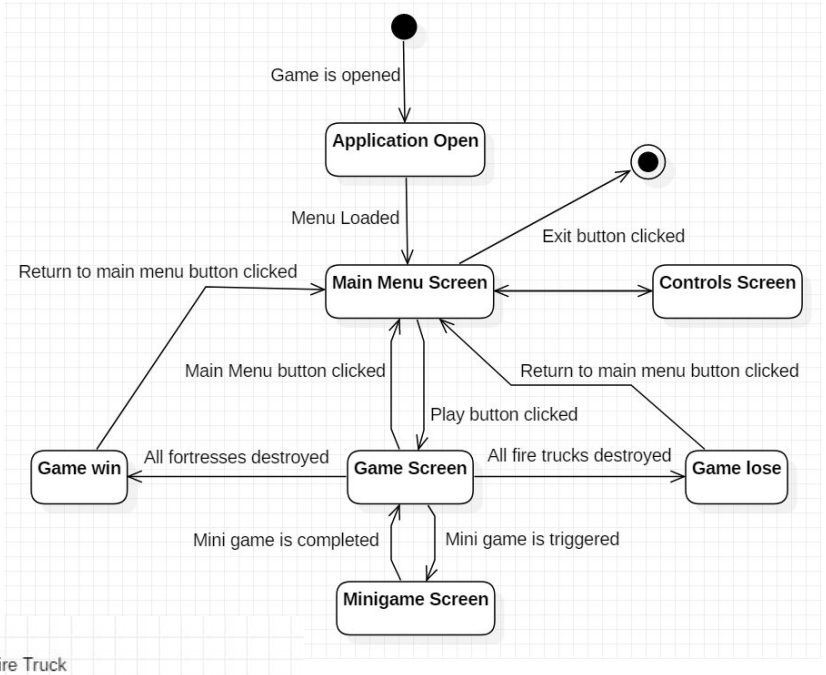


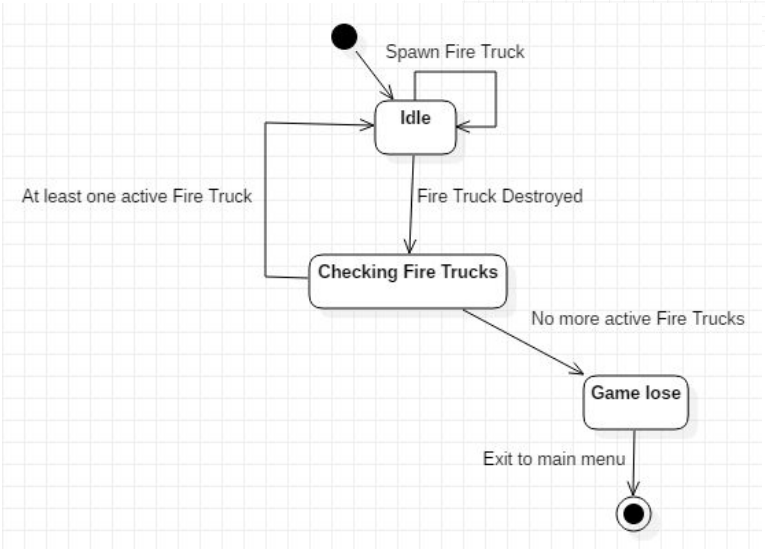
**AP stands for Attack points, HP stands for Health points*

State Diagrams

Application Overview

This state diagram shows the general progression the user will experience through the application, starting from when the application is opened to when the application is closed (**UR_LOSE, UR_WIN, UR_MINI_GAME, FR_CONTROLS**)



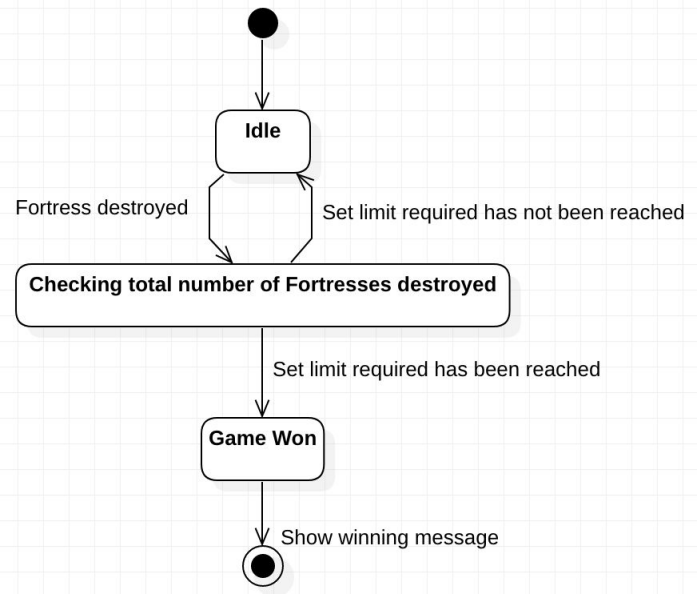


Fire Truck Destroyed

This state diagram shows the process of what happens when a fire truck is destroyed (**FR_ATTACK**). This also forms part of a user requirement (**UR_LOSE**) which describes that the game ends when all fire trucks have been destroyed.

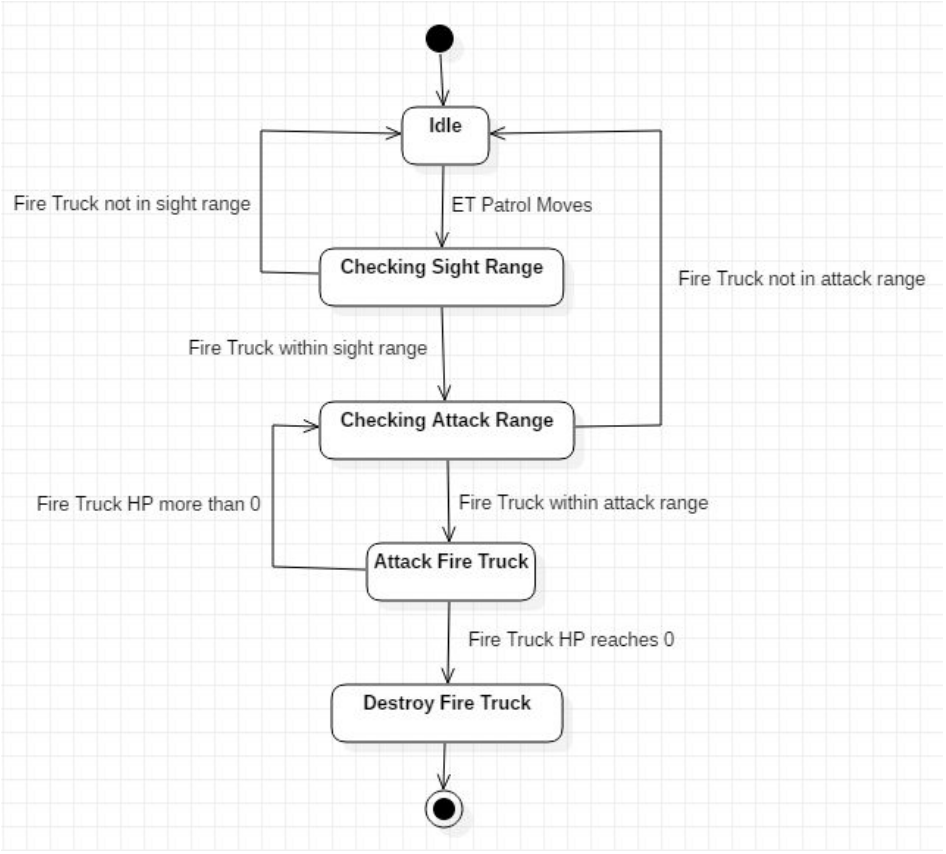
Fortress Destroyed

This state diagram shows the process of what happens when an enemy Fortress is destroyed by the player. Enemy Fortresses will keep spawning stronger until the player destroys enough to reach a set number required to win which will be decided during implementation. (**UR_WIN**)



ET Patrols

This state diagram shows how the ET Patrols interact with the Fire Trucks. These patrols can chase fire trucks when they are within a certain range (**FR_PATROL_SIGHT)** and then attack the fire truck when they get closer **(FR_PATROL_ATTACK)**. When the fire truck's HP reaches zero, it is destroyed.



Explanation and Justification

Entity: *Entity* is an abstract class encapsulating all interactable objects within the game. Each entity has an *x* and *y* attribute corresponding to its map coordinates. We have excluded a *destroy()* method as it should be assumed that every object can be destroyed as well as created.

Fortress: The *Fortress* entities are immobile (**FR_MOBILITY**) enemy bases that the player aims to destroy. Each *Fortress* has a set value for its *HP*, *range* and *AP* for attack points (amount of damage they inflict on fire trucks). Upon instantiation, the Fortress object is passed an integer for *type,* effectively representing a stat multiplier. As the game progresses, *type* increments thereby increasing the *AP* and *HP* of spawning Fortresses making them harder to flood **(FR_WATER)**. The *attack()* method subtracts the Fortress' *AP* from the *HP* of *FireTrucks* within the fortresses range (**FR_FORTRESS**).

Station: The *Station* class extends *Entity* and serves as the *FireTruck* spawner and repair/refill center as instructed by the brief ("*Fire Engines need to return to the Fire Station to repair and refill*"), hence it contains the methods *spawnFireTruck()*, *refill()* and *repair()* (**UR_REPAIR**). The association between the *Station* and *FireTruck* class shows that the Fire Station spawns 4 or more fire trucks (**UR_FIRE_TRUCKS**).

FireTruck: The *FireTruck* entity is controlled by the player. *AP , reserve, range, speed, HP* and *type* all play a role in creating a unique specification for each fire truck (**FR_FIRE_TRUCKS**). The *move()* method allows for the object to change coordinates based on user input (**FR_MOBILITY**). When in range of a Fortress, the *attack()* method gradually subtracts its *AP* from the target Fortress' *HP* based on user input. However, our client said that fire trucks cannot harm ET patrols.

Patrol: *Patrol*s are indestructible entities moving around either pre-defined routes or "smartly-generated" ones (**FR_MOBILITY**). Each *Patrol* can attack *FireTruck entities*, and would deal damage to it depending on the value of the *AP* attribute. Patrols will chase fire trucks that they are within their line of sight defined by *sightRange* and attack the fire trucks when they are within the patrol's attack range defined by the *attackRange* attribute. (**FR_PATROL_VIEW_TRUCKS**).

Screen: The abstract class *Screen* efficiently loads and clears the memory of graphics and objects only needed in specific contexts. It contains the methods *create()*, *render()* and *dispose()* ; *create()* is the first method to be called when an instance of a screen is made and is used to set up all of the main elements on that screen. *render()* updates every element that is prone to change. *dispose()* is called when the screen is no longer needed and clears the memory of anything that was created in that screen, as well as the screen itself. The subclasses *MenuScreen*, *ControlsScreen*, *GameScreen*, and *MiniGameScreen* inherit and implement these methods.

MenuScreen: This screen is created upon opening the application. From here, the user is able to go to the Controls Screen with the *toControls()* method and the Game Screen with *toGame()*. The user also returns here after the game has finished.

ControlsScreen: The controls screen shows the user how to play the game (controls/objective) (**FR_CONTROLS**). Users can only access it from, and go back to the main menu screen hence the *toMenu()* method.

GameScreen: This is where the user will play the game. Preliminaries such as creating the map and an instance of *Game* (stores information about the progress of the player) happen in the *create()* method (inherited from the *Screen* class). Every action/event during the game happens inside the *render()* method (updated every frame). To support the mini-game, our architecture the methods *pause()* and *resume()* which temporarily stops the *render()* method from updating so that the progress of the main game is not affected while the player is completing the minigame (i.e. patrols damaging fire trucks). Once the minigame is completed, the main game is resumed. Alike the other screens, to

go to the minigame screen, the *toMiniGame()* method is called. The *pause()* method can also be used to show an overlay display, which will pause the game and give the player the ability to view controls, exit to the main menu or *resume()* the game.

MiniGameScreen: We decided that the mini-game will have its own screen to avoid confusing elements of the main game with those of the mini-game, as well as efficiently disposing of temporary resources specific to the mini game (**UR_MINI_GAME_THEME**). Once the mini-game has concluded, the *toGame()* method is called to return to the game screen.

Game: The *Game* class is a singleton class meaning there is only one instance of it at any time. Its purpose is to keep track of all game data required for it to operate as requested while *GameScreen* runs the game elements. *fortressesDestroyed* is incremented for every Fortress the player floods, and serves as the principal progress tracker. Once this equals the set number required for the game to be won, gameWin() is called which tells the user that they have won (**UR_WIN**). Fortresses *type* will increase accordingly and more Patrol entities will spawn (**FR_INCREASE_PATROLS**). *activeFireTrucks* keeps track of the number of FireTruck entities on screen. If this number reaches zero, *gameLose()* is called which tells the user that they have lost (**UR_LOSE**). *startCountdown()* starts a countdown to zero, once reached it calls *destroyStation()* destroying the fire station (after which the player can neither repair nor refill its Fire Trucks) (**UR_STATION_DESTROY**). Fortresses grow stronger as the user floods them, focusing on the need for strategy, while the countdown focuses on the need for speed (**UR_ENJOYABILITY**).