Module	SEPR
Year	2019/20
Assessment	3
Team	Early Bird
Members	James Little, Ryan Vint, Adam Lynch, Georgina Martin, Kheng Yeoh, Tanay Malde
Deliverable	Implementation Report

Changes to previous software:

Change	**Items Affected**	**Justification**
Particles able to attack all entities	Particle, Patrol, Fortress, Firetruck	Patrols and fire trucks were required to be able to attack each other. Therefore, in order to reuse the particle class it had to be changed to work with all entities as opposed to only fire trucks and fortresses.
Method to locate nearest fire truck to a particular tile	FireStation	In order for patrols to target the nearest fire truck when attacking and following them. Must therefore also pass in the game screen in order to use the obstacle grid when pathfinding.
Both fire truck and fortress now extend entity	Fire truck, Fortress	To make code more concise as well as create a code structure that allows one to iterate over entities.

Additions for implementation of requirements:

Change	Items Affected	Justification
Implemented FR_WATER requirement	Additions to GameScreen class and fortress class	As stated by the FR_WATER requirement, the alien fortresses must take more water to destroy over time. We added this functionality by having a downwards counting timer which, when it hit 0, increased the health of the fortress (thus the amount of water it takes to destroy them) in addition to increasing damage, attack rate and range to increase the difficulty of the game.
Implemented an entity class	Entity, Fire Truck, Unit, Fortress	Needed a class to hold parameters such a position, HP and game Screen which were common to many classes.
Implemented a unit class	Unit, Patrol	Needed a unit class to set out basic movement functionality. Units have a queue of Vectors corresponding to tile on the map and a queue of points which is used to hold vectors coordinates of points between tiles. When a unit completes all moves in its points queues it then refreshes it with

		the vectors corresponding with movement to the next tile.
Implemented Patrol section for FR_AI requirement	Patrol	Patrols have rudimentary AI, after chasing a fire truck and going off course they will path towards their next node using an A* algorithm.
Implemented multiple types of patrols	Gunner, Bomber	Added two different types of patrols, one which attacks from a distance and another which attempts to close the gap between itself and a nearby fire truck before self-destructing to damage it.
Implemented FR_VIEW_TIMER requirement	Additions to GameScreen class	Added the functionality of a downwards counting timer on the main game screen which counts down to the time of which the fire station will be destroyed thus allowing no more repairs to fire trucks. This fills the requirement of FR_VIEW_TIMER.
Implemented FR_PATROL_INCREASE requirement	GameScreen	When a fortress is destroyed, a patrol is spawned on the nearest tile below it and the maximum number of patrols is also incremented. This ensures that as the player progresses in the game the number of patrols increases.
Implemented patrol spawning	GameScreen, Patrol	When patrols are destroyed, new ones need to be created in order to maintain difficulty. Therefore, as long as there are less than the max number of patrols a new one is spawned every 10 seconds.
Implemented FR_PATROL_DAMAGE and FR_PATROL_ATTACK requirements	Gunner, Bomber	Each patrol fulfills the requirement differently. The gunner will attack the nearest patrol within its range by using the same particle system that the firetruck uses, this inflicts less damage than the bombs of the fortress. The bomber, which also deals damage through its attack function, inflicts a large amount of damage when it is within a tile of a fire truck.
Implemented FR_PATROL_SIGHT requirement	Gunner, Bomber	Patrols have a range parameter. For the gunner, this directly affects the range of its attack. However, the bomber targets trucks within its viewDistance and moves towards them. This implementation fulfills the FR_PATROL_SIGHT requirement.
Implemented FR_ACCESS_M	Additions to GameScreen class and creation	To fulfill the requirement of FR_ACCESS_MINIGAME we made it so that once the final fire station is destroyed the screen

INIGAME requirement	of MiniGameScreen class	switches to the newly added MiniGameScreen class which runs the minigame.
Implemented FR_STATION_DESTROY requirement	Additions to GameScreen class	FR_STATION_DESTROY requires that fire trucks cannot be repaired or refilled when the fire station is destroyed, this is implemented through removal of the fire station entity in GameScreen no longer allowing it to use any of its functionalities.

State Diagrams

We have implemented the state diagrams from assessment 1 architecture [1] and tested these to make sure that some of the originally designed game systems run as intended. The state diagram for application overview is tested by MAN_MINIGAME_WIN and MAN_MINIGAME_LOSS_NOENGINES from assessment 3 testing [2] and MAN_CONTROLS from assessment 2 testing [3]. The assessment 3 tests show full run throughs of the game from start to either loss or victory including playing the minigame hitting all of the states in the state diagram apart from the controls screen, and the assessment 2 test is based around the movement between the main menu and controls screen meaning we have tested all states and transitions between them for this diagram.

The fire truck destroyed state diagram is tested by MAN_GAME_OVER_LOSE from assessment 2. In this test the game starts, all fire engines are killed, and once they are all dead the user is taken to a game over screen from which they can return to the main menu. This test runs through all states of this state diagram.

The fortress destroyed state diagram has been altered slightly to allow for our minigame implementation as our minigame launches when the final fortress is defeated, there is now an extra state between ‘checking total number of fortresses destroyed’ and ‘game won’ that plays the minigame, and moves onto win if the minigame is won. This update is necessary for our design of the minigame as a final boss battle otherwise the minigame would have to launch in the middle of the game, no longer making it feel like a final boss that the player must defeat, which was our aim when designing the minigame. Having this feeling of a final boss adds to the enjoyment of the game that, according to user requirement UR_ENJOYABITILY, must be sufficient for the game’s target audience.

The ET patrols state diagram has been implemented and tested as shown by the assessment 3 tests MAN_PATROL_FOLLOW and MAN_PATROL_ATTACK. MAN_PATROL_FOLLOW shows the first stages of the state diagram, of the patrols identifying fire trucks in their sight range and then stopping trying to attack them when they are out of sight range and going back to standard patrolling. MAN_PATROL_ATTACK shows the implementation of the patrols attacking and dealing damage and eventually destroying the fire trucks, completing the state diagram.

Minigame Implementation

One of the largest parts of implementation required for assessment 3 was the minigame. We decided to implement a turn based final boss battle between the fire truck and the King of Kroy. For this we needed to implement a new Screen and a new Event Input Handler that allows the user to select different moves and attack the alien.

If the player defeats the alien they are taken to a game win screen. If they lose however they are returned to the main game, with the attacking fire truck destroyed, and the final fortress respawned with low health to allow the player to attempt again to defeat the boss and reward them for having more fire trucks alive at the end of the game by giving them extra tries at the minigame.

A* algorithm

In order to efficiently move units across our map while ensuring they move along the roads on our map we needed to implement a pathfinding algorithm. A* enables us to find the shortest path between any two tiles on our map while avoiding obstacles which are stored in an obstacle map in our game screen. It creates a map of nodes and assigns a weight to each of these nodes equal to the weight of the distance from the start plus the estimated distance between that node and the finish. It then iterates through these nodes to find the shortest path between start and finish. We needed to use this algorithm when moving patrols across our map as we wanted patrol paths to be randomly generated. By randomly generating nodes which the patrols would path between we ensured that their movement was complex while not needing to hard program every path thus creating more replayability in the long run.

References

[1] Architecture - Assessment 1 [Online] Available: https://gm-martin.github.io/assessment1/Arch1.pdf [Accessed: 16 Feb. 2020]
[2] End to end tests - Assessment 3 [Online] Available: https://gm-martin.github.io/assessment3/EndToEndTests.pdf [Accessed: 16 Feb. 2020]
[3] Manual Tests - Assessment 2 [Online] Available: https://gm-martin.github.io/assessment2/ManualTests.pdf [Accessed: 16 Feb. 2020]