Implementation Report

Mozzarella Bytes

Assessment N°4

Kathryn Dale

Daniel Benison

Elizabeth Hodges

Callum Marsden

Emilien Bevierre

Ravinder Dosanjh

* Changes to the code are indicated in the codebase by a banner above the class or method that has been added or modified. The figures referred to below can be found at [1].

Changes to inherited software

Changes to the software that we inherited: Before, the code for a button was included in each screen class leading to repeated code as multiple screens use the same buttons. To rectify this a *Button* class containing the home, pause, sound, information and save buttons and an interface *ButtonBar* (providing the methods a screen will have to override to use these buttons), was created. *MenuButton* and *MenuToggleButton* were also created and contain the code for the menu screen buttons.

Previously, the boss patrol was created at the start of the game even though it does not destroy the fire station UR_DESTROY_STATION until a number of fortresses are destroyed. This led to a lot of if statements in *GameScreen* to check whether a patrol was a boss patrol making the code hard to read. As the boss patrol has different functionality to the other patrols we created a *BossPatrol* class that extends *Patrol*. This improves code readability and allows the boss patrol to only be created when needed.

Before, the player could pause the game at any point and spend any length of time drawing the trucks' path. To incentivise the player to use this feature and to integrate it into the gameplay more we created a freeze *PlayState* where we limit the time the player can spend in the freeze screen and only allow the player to access this feature (by pressing the spacebar) after a length of time has passed. To enable the player to pause the game for an indefinite amount of time we created a pause *PlayState* (accessed by pressing the pause button).

Changes to the GUI that we inherited: As *GameScreen* and the minigame screen (*DanceScreen*) implement *ButtonBar* both screens now have buttons in the top right hand corner which the player can use to return to the home screen, toggle the sound (FR_SOUND_OFF), pause (FR_MINIGAME_PAUSE) and save the game (UR_SAVE), or view the controls screen (FR_CONTROLS). For UR_ENJOYABILITY new background images for the minigame, game over and menu screen were added and the music for the minigame was changed to a more up tempo song to increase the pace of the gameplay. The screen also has a different coloured overlay to differentiate between when the game is paused or frozen.

How the software was modified to incorporate the changes to the requirements

Difficulty level GUI (see fig.4): From the new *DifficultyScreen* the player can click on either the easy, medium or hard button to select a difficulty level (FR_DIFFICULTY_SELECTION, FR_MENU). To incentivise the player to play each level, each difficulty level has a different map.

Difficulty level code: The three difficulty levels (easy, medium & hard FR_DIFFICULTY_LEVELS) have a different (UR_DIFFICULTY): tile map, time until a new power up is spawned, number of fortresses that need to be destroyed before the fire station is destroyed, number of patrols, time until a freeze becomes available, AP, HP, range and speed for each fire truck, initial AP and HP for each fortress and time until the fortresses AP and HP increase (see [2]). These values are stored in an enum *difficultyLevel*, making it easy to change these values to help balance the game (UR_PLAYABLE) and to add new difficulty levels. To implement these values in *GameScreen*, the *difficultyLevel* the player chose from *DifficultyScreen (*by clicking on the easy, medium or hard button) is passed into *GameScreen's* constructor where the previously hardcoded values mentioned above are replaced by the values from the selected *difficultyLevel* (see enum DifficultyLevel and GameScreen lines 176-207).

Saves code (see fig 1): To save the game from *GameScreen* (UR_SAVE) the player presses the save button, this calls *saveGameState*. Each entity has a method *getDescriptor* which creates and returns an instance of its associated static class (nested in the static outer class Descriptor (Desc)) containing the values that need to be saved for each entity (the entity attributes in the diagram). These classes are static to provide a layer of abstraction clearly showing which attributes need to be saved for each entity and because no objects need to be initialised (since these values are going to be written to a JSON file). These values are added to *saveMap* alongside other values needed to restore the game to this state. To save from the minigame (FR_MINIGAME_SAVE, UR_SAVE) the player presses the save button which calls *saveGameState* from *DanceScreen*. This calls *savegameFromMinigame* passing in the static instances of the patrol and firetruck participating in the dance off and adds these to *saveMap* before calling *saveGameState. saveGameState* writes *saveMap* to a JSON file which is saved to the assets/saves directory. As multiple JSON files can be written and stored in this directory the game can support multiple saves (FR_NUMBER_OF_SAVES).

To restore the game (see fig 2, FR_SELECTING_SAVES) *SaveScreen* creates an instance of *SavedElement* for each JSON file. *SavedElement* instantiates each saved object by overloading the object's constructor with the values from the saved file. This is done for each saved file as some of the values are rendered to *saveScreen* to help the player remember the games they saved. The chosen game (*savedElement*) is passed into *GameScreen's* overloaded constructor where the saved entities are retrieved from *savedElement*. As only one GameScreen was needed it was not initialised for each *savedElement,* hence to assign the GameScreen in firestation *setGameScreen* is called.

Saves GUI: The player can access *SaveScreen* from the *MenuScreen* (see fig 5, FR_MENU). To display an unlimited number of saved games (FR_NUMBER_OF_SAVES) on the screen without affecting the formatting a scroll pane was used. From the scroll plane the player can view: a thumbnail of the game (taken by *takeScreenShot* when the game was saved), the time and date it was saved, how many trucks, fortresses and patrols are remaining and whether the FireStation is alive. These values were chosen to help remind the player of the state of each game. The player can select a saved game from the scroll plane to see more stats and a larger screenshot of the game and either start the game by clicking the start button (FR_SELECTING_SAVES) or delete it (FR_DELETING_SAVES) by clicking the delete button.

Power ups GUI: There are five power ups (UR_POWER_UPS) which allow the player to: freeze the game (*freeze),* restore a truck's HP (*heart*), temporarily increase the truck's range (*range*), restore a truck's reserve (*water*) or temporarily make a truck immune to attacks by patrols and fortresses (*shield*). To help the player easily spot and identify available power ups we used the LibGdx animation class to make the power up spin and gave each power up a unique Texture. The player has a certain amount of time (*timeOnScreen*) to collect a power up before it disappears (FR_POWER_UPS_TIMEOUT) which can be seen by the bar rendered above each power up showing its *timeLeftOfScreen*. When a power up is clicked on this, alongside the effect of the power up, is shown in the top right hand corner.

Power ups code (see fig.3): At regular time intervals (dependent on difficulty level) a LibGdx Timer in *GameScreen* calls the method *canCreatePowerUp*. This method randomly chooses a location from the map's road tiles (extracted from the tile map by *generatePowerUpLocations* and stored in *powerUpLocations*) before calling *createPowerUp*. *createPowerUp* spawns a random power up only if no other powerup shares this location; this stops powerups overlapping (FR_POWER_UPS_OVERLAP). Each power up extends the abstract class *PowerUps* where it overrides the *invokePower* method. *invokePower* is called when a truck and a power up are on the same tile triggering the effect of the power up; multiple *invokePower* methods can be called at the same time (FR_POWER_UP_STACK). The design decision to use polymorphism makes the code more extendable as a new power up can be easily added by creating a new class which extends *powerUp* and overrides *invokePower*. Polymorphism also makes it possible to iterate through, and update, the power ups without needing to know the specific type and behaviour of each power up thus improving the codes readability and extendibility.

Additional changes

Additional changes to software: in the freeze screen the player can undo and redo parts of the trucks' path. To implement this the trucks path is split into segments, a segment contains the route from when a player clicked on a truck (or the end of a trucks segment) to when they clicked up, each segment is stored in a queue. Each of these queues is stored in another queue *pathSegments* which contains the trucks entire route. When the player presses undo the last drawn segment in the trucks route is popped from *pathSegments* (removed from the trucks route) and pushed to *pathSegmentsStacks*; when the player presses redo the segment at the top of *pathSegmentsStack* is popped and pushed back onto the truck's *pathSegments* (added to the trucks route).

Additional changes to GUI: For UR_ENJOYABILITY visual effects (*VFX*) were added. *VFX* uses the LibGDX animation method and is called when a fortress is destroyed (FR_ANIMATION) creating a visual explosion.

Reference List

1 Mozzarella Bytes, *Implementation diagrams,* 28.04.20. [Online] Available at: https://db1218.github.io/KroyAssessment4/assessment4/overflow/Implementation%20diagrams.pdf

2 Mozzarella Bytes, *Difference between difficulty levels,* 28.04.20. [Online] Available at: https://db1218.github.io/KroyAssessment4/assessment4/overflow/Difference%20between%20difficulty%20levels.pdf