Module	SEPR
Year	2019/20
Assessment	4
Team	Dalai Java
Members	Jack Kershaw, Max Lloyd, James Hau, Yuqing Gong, William Marr, Peter Clark.
Deliverable	Implementation

For this assessment we were provided with 3 small changes that the client required to be implemented in our version of the game KROY. These are the addition of powerups, a difficulty selection as well as the ability to save the game state and continue on from that point later on.

From the code that we inherited we didn't need to perform any large refactorings to the architecture in order to implement these features. We did however add a variety of methods, classes as well as the inclusion of the gson object serialization library imported via gradle.

Within the PlayState class we implemented a new constructor that takes an additional parameter of a JSON file containing the textual representation of the various sprite classes that contain all the required values to recreate the state of the game at the instance that it was saved. We took the decision to not save the position of the various projectiles and bullets on the map as it would have a minor effect on the save game.

Two new states were required as well, LoadSate and SaveState. Both are simple menu states that allow the user to choose which save slot should be used before either loading or writing the object respectively.

Creating Different Difficulty Modes

Our implementation allows the user to change the difficulty between three different modes: easy, normal and hard. Easy halves the amount of health that the fortress has making it considerably easier to kill, normal is as the game was before. Hard has an increase in total health of the fortress as well as the ETs being considerably more deadly now. The ETs both shoot more often as well as each projectile doing more damage to the player. All of these changes are performed using a simple difficulty multiplier defined inside the Kroy Class to enable all classes and methods to be able to access this value. The values we decided upon for this was 0.5 for easy, 1 for normal and 2 for hard(FR_change_difficulty). Fortunately, no major changes were required to the the architecture in order to implement this just simply adding a multiplier to certain values in object declaration

A key thought for us was ensuring that changing the difficulty was extremely straightforward and could be done in a place that makes the most sense. Because of this we decided upon putting the difficulty on the level select screen(FR_select_difficulty). This ensures that the player is fully aware that this option exists as if it was hidden in the setting screen many players may not even realise that this feature has been implemented.

As for how the user chooses the difficulty we decided upon using tick boxes that are clearly visible to the user when choosing which level to play. Other thoughts included using a drop down menu to display the options as well as sliders that are also seen in a large variety of other games. The tick boxes were chosen as there was only a limited number of options and plenty of space it would make the users life the simplest as all the options are clearly laid out in front of them. This would prevent them not knowing if more difficulty levels exist removing any possible confusion that this could arise from this.

Adding Power Ups to the Map

Every 10 seconds a random power up is added to the map in a random location. There are 5 different power ups that can be added to the map(FR_power_ups). These are infinite health, a permanent health increase, increased range, increased speed and increased damage.

When 10 seconds has passed, the PlayStates update method will first decide upon which type should be used before choosing where upon the map it should go. However, as on the map some tiles are collidable and so the firetruck can't reach them. If the power up is to spawn on one of these tiles a new location is found until it is in a valid location.

When a fire truck collides with one of these the trucks powerUp() method is called. This takes a string of the power ups type in order for the required effect to take place. The effects that only last a specified amount of time work by using Timer.task() a form of concurrency in java that implements the runnable interface. After the specified period of time the effect is reversed in order to make the powerups only a temporary advantage to the player.

Implementing a Save System

To allow for this to function correctly we took advantage of the GSON library that allows for easy serialization of objects within java. Using these serialized objects we then formatted them into a json file so that they were both easy to read and more importantly allowed us to restore the state of the game later on when requested by the player. A new save game is created when the user clicks the exit game button but not when they just quit the level. We decided upon this as we felt that if the player is choosing a different level they have made a conscious choice to stop playing that level and so wouldn't benefit from a save file. Whereas if the whole game is exited(UR_save_load_quit), it may be because they have run out of time or have more important things that need doing.

As a group we also decided upon not saving the projectiles from either entities as we thought that it would unnecessarily overcomplicate the save file as well as from playing over released games where projectiles and power ups are usually not saved anyway.

Loading the game is achieved by choosing the "load game" button on the main menu which takes you to a new state which will present you with the three most recent saved games. Each of these buttons creates a new start state which takes advantage of the second constructor that was implemented for this assessment. It takes an additional parameter, the save file. This is similar except that rather than using the default values for each level it parses the data from the save file. To achieve this the Simple Json library was used as it appeared to be the most lightweight approach to this. Initially we tried to use GSONs deserialize method to do the same thing but an issue arose due to how the texture would be loaded in. Rather than the file name like required by the Texture class in libGDX, it parses the actual data from that file. An initial approach to overcome this was to create new abstract classes that only contained the attributes that varied depending on the save but this turned out to be more complex than we expected and became a lot of work to implement so in the end just settled for the manual approach.