SEPR

SpaceKey Projects

Assessment 2

Architecture

Report

JORDAN CHARLES
SAMUEL HUTCHINGS
CHLOE HODGSON
GOLNAR KAVIANI
TAMOUR ALTAF
JACK THOO-TINSLEY

2019/2020

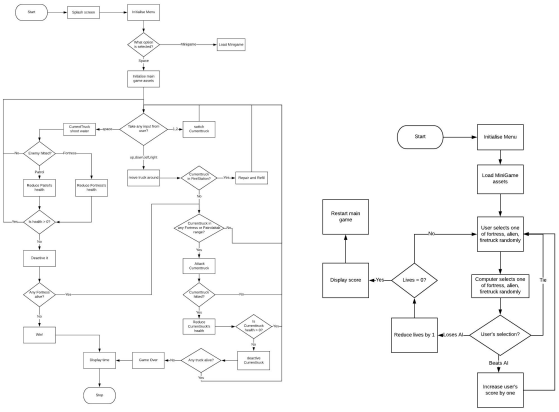
Architecture

a) Abstract Representation

We have supplied two flowcharts describing the user flow, and a UML diagram for an abstract description of classes below to help us have a better understanding of the game and will give us an idea of how to program our different classes. Please note these are more easily viewable on our website.

Flowcharts

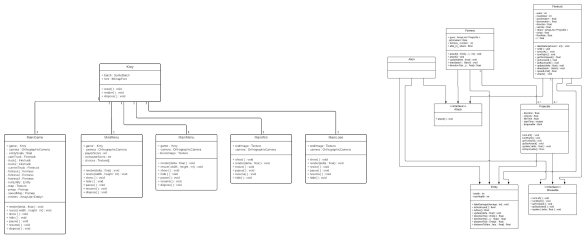
We have changed some parts of the flowchart in comparison to our first assessment, because as we wrote the code we realised we needed to change some parts to make our code more efficient. We tried to add more detail in the flowchart as we figured out this would help others understand our game better. We have now designed two flowcharts: one describes the main game and the other one describes the mini game.



The first flowchart is what describes the main game. When the game is opened, our team logo is displayed as a splash screen, and then the main menu is initialised. This menu tells the user how to play the game, and will also give them the option to either load the main game or load the minigame. If the minigame is selected, we load the minigame (the user flow for the minigame is described in another flowchart). Otherwise, we load the main game assets (libGDX requires them to be loaded in memory before we can even start the game). From here is where the main game has started. We check if we have received any input from the user. If the user presses space, then the current truck we are controlling will shoot water from its water cannons. If a patrol or a fortress is hit by the water, we lower its health. If its health becomes zero, we eliminate it from the game. We then check if there are no more fortresses alive. If there aren't we win (which displays a "you win" image onto the screen, then stops the game). Otherwise, we check if the current truck we control is in the attack range of a fortress or a patrol. We also reach this point if, after the user presses an arrow key (which moves the current truck in the respective direction), the current truck is not in the range of the fire station (if it is in the range of the fire station, we fill up both its health and water at a constant rate). If there is a truck in range, we attack it. If our attack hits our current truck, we lower its health, and then we deactivate it if its health becomes zero. Finally, we check if there are any trucks alive, and if not we lose the game (which displays a "you lose"

image and stops the game). Whether we win or lose, the user flow is the same from here. We display the user's score (not implemented yet) which is calculated by how efficiently we played the game (not being hit and quickly disposing of fortresses and patrols increases our score much faster). After scores are displayed, we end the game (which closes it).

The second flowchart describes how the minigame is played. When it starts, we initialise the menu and load its assets. The game instructs the user to select an option to put against the AI (akin to rock paper scissors, except they relate to the style of Kroy). The AI will then pick a random option of the three. Depending on the user's selection: winning will grant them a point, losing will reduce their lives by one, and tying will do nothing. If the user ls a life, we check whether their lives are now currently zero. If they are zero, then we display their final score, and restart from the main game (thw Kroy main menu). Otherwise, we instruct the user to select one of their choices again.
UML Class Diagrams



The UML class diagram is an abstract representation of how our game objects relate and interact with each other. We created our classes by systematically going through the brief and our requirements, and deciding what unique classes we'd have to create to meet our needs. The first step was to figure out the names of every unique thing that might need to be created, such as the fortress, aliens, and the fire station. For some of these objects, we grouped them into what they might share in common given what was described in the brief, such as how the station, fortress, aliens, and firetrucks all have health and a position, but the station does not need to attack like the other classes do, and the fortress also does not need to move around. Once we figured these out, we created methods and attributes which described what we could do to each of these classes, and what information describes them. Finally, we added some associations, so we knew how exactly the classes related to each other.
When it came to looking at our architecture, we decided that it may be better to split it into two seperate UML class diagrams. One defines the classes we use in our main Kroy game (e.g Fortresses, FireTrucks, Projectiles), and the other defines our menu screens that we call depending on what state of the game we are in (MainMenu is our screen that tells the user how to play, MenuWin is the screen that displays when the user has won the game).

b) <u>Systematic Justification</u>

In the below table, we have described the classes present in the class diagram. We have explained why we have created our classes in this way, what each class includes, and how they satisfy user requirements.

Entity	Entity is the base class that all our game objects inherit from. Because many of our classes share the same attributes and methods (like health), it makes

	more sense to define these in a parent class and then encapsulate them there instead of having to redefine them in each child class. Because every entity has a health value, this helps meet our FR_ENEMY_HEALTH and UR_HEALTH requirements, which require the health status of our entities to our user. We decided if we Entity inherits from Sprite, which is a predefined class in Libgdx, we do not need the Coordinate class as we can just pass our texture in as method arguments and access the coordinate by using getPos(). takeDamage() reduces the health of Entity depending on the value we pass in. The isAlive() method checks if the health of Entity is 0 or not. Because we have an isDestroyed() method, we can check if something is dead but still keep it alive in our game world (i.e we do not delete the entity). So if firetrucks or the fire station are destroyed, we can still leave their sprites in game but render them unusable. We also defined methods directionTo( ) and distanceTo( ) here to make it easier to locate other entities especially for calculating attack ranges in Entity's subclasses.
Projectile	This class is a subclass of Entity. It inherits from the Attack interface. As both Fortress and Firetruck (and Alien in future) use the attack() and we needed to define an ArrayList to keep track of bullets (their positions, and when we should draw them on the screen), we decided instead of other classes having to figure out how to make their own projectiles, we created the Projectile class. isDisposable() decides when we should remove the bullet from the screen which is either when bullets have ran out of lifeTime or they hit an object.
Fortress	This class is a subclass of Entity. While it doesn't contain any unique methods or attributes, it is an entity that must be able to attack, so having it be its own unique class that inherits from the attacker interface means that when we implement the attack() method, by adding an attribute fortress_number and passing it to the attack() method we can have it unique to each fortress (instead of having a generic attack that all fortresses share) which is needed to meet FR_VARIATION. This class uses an attribute goos as an ArrayList<Projectile> for having a count of bullets shot. Fortresses should only be able to shoot one at a time so have an attribute able_to_attack. Whenever we remove bullets from goos (because of hitting an object or running out of lifeTime) then if any FireTruck is still in attack range they can attack again. The multiplicity associated with Fortress and Alien represents that each fortress can spawn one or more alien patrols, which are "attached" to that parent fortress. We also defined the directionTo() method which calculates the direction from the fortress to an entity.
Firetruck	Firetruck is a subclass of Entity. It inherits from the Attack interface, but as it can move it also inherits from the movable interface. Uniquely, it has a water attribute that represents how much water the truck has, and methods to represent accessing its water level, and refilling it. We also define maxWater here, as we dedicated different capacities of water for each of our Fire trucks. For attack(), Firetruck has an attribute drops of type ArrayList<Projectile>. We can attack up to 50 shots. Despite the last assessment, we decided we do not need a Station class, as all the methods related to Station can be easily defined in the Firetruck class so instead of

	having a class we just define an area on the map as the station, so if a truck goes there, it would get repairs and refills. FireTrucks are coded to have more speed when they are driving on roads in comparison with when they are on grass, so in order to get different speeds we defined acceleration and deceleration which contain vectors of how quickly to change speed. We also defined speedLimit() so they can not go higher than the permitted speed at each place. Method draw() overrides the original sprite draw method to draw water drops simultaneously.
Alien	Alien is a subclass of Entity, which implements the Movable and Attack interfaces. Instances of Alien are only spawned by fortresses, so there is a method and attribute available to get the Fortress object that created the Alien. This is good for if we want to include any additional functionality that we haven't deemed necessary in our requirements, such as aliens of a certain fortress homing in on any nearby fire trucks if they are attacking it.
Attack	Multiple classes require different ways to attack. We could have created a parent class for them to inherit from, and making this class abstract would have ensured you couldn't have instantiated an object of this bass class. However, because we only need to define an attack() method, and no attributes, making this class an interface was better suited for our needs. Each class that implements attacker will have its own custom definition of the attack() method. Fortress in particular will require many different unique attacks at the implementation level in order to meet our FR_VARIATION requirement.
Moveable	Similar to Attack, it was better suited to use an interface instead of an abstract class. When we implement the +movePos() method in the Alien class, movement may be more random or perhaps alien patrols will slightly home in onto the nearest fire truck. The implementation in FireTruck however will instead need us to ensure that when an arrow key is pressed (we use arrow keys because FR_CONTROLS defines we must use keyboard input), the fire truck that moves is only the fire truck that we are currently controlling.
MainMenu	We designed a page for the beginning of the game. Which gives us a total information about how we can play the game(like which keys can we press if we want to move a truck). We need to press "space" key in order the game.
MainWin	If the user manages to kill all the Fortresses(and also Aliens in future) the game finishes and a win page would come on the screen to celebrate a users win.
MainLose	If all the trucks get destroyed, user would lose the game and this would be to tell the user by a loose page coming to the screen. At this point the user can either try the game again or exit the game.
Kroy	While no class inherits from Kroy, it acts like a master class as it is the first class we call in our MainGame and it basically creates the batch screen and basics of the game.