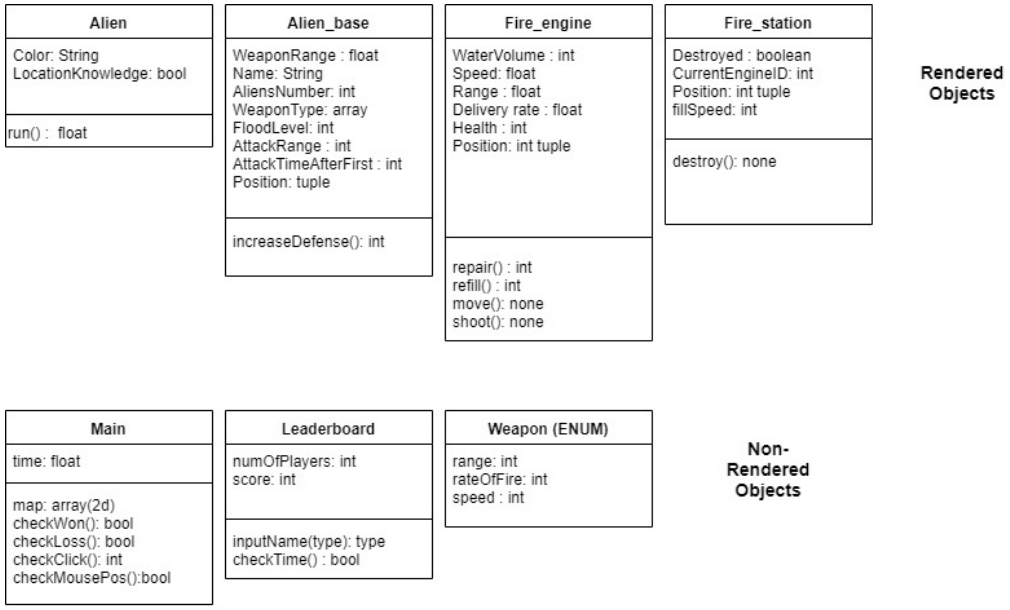
MiKroysoft

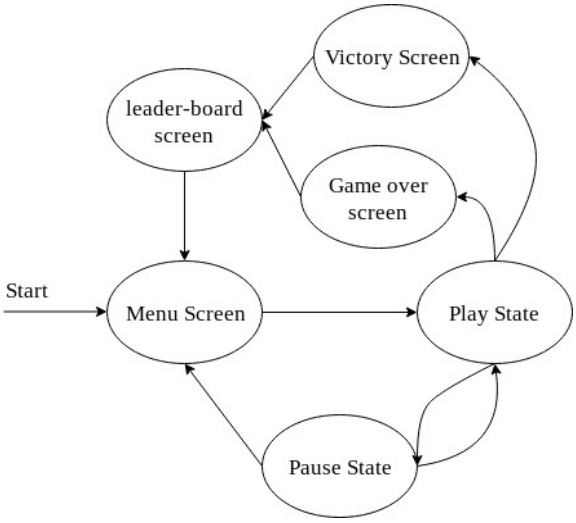
Module	SEPR
Year	2019/20
Assessment	1
Team	MiKroysoft
Members	Daniel Crooks, James Rand, Irene Sarigu, Alfie Jennings, Charlotte Clark, Jasper Law
Deliverable	Architecture
Website	https://mikroysoft.github.io/

3A

Main classes diagram (Figure 1)



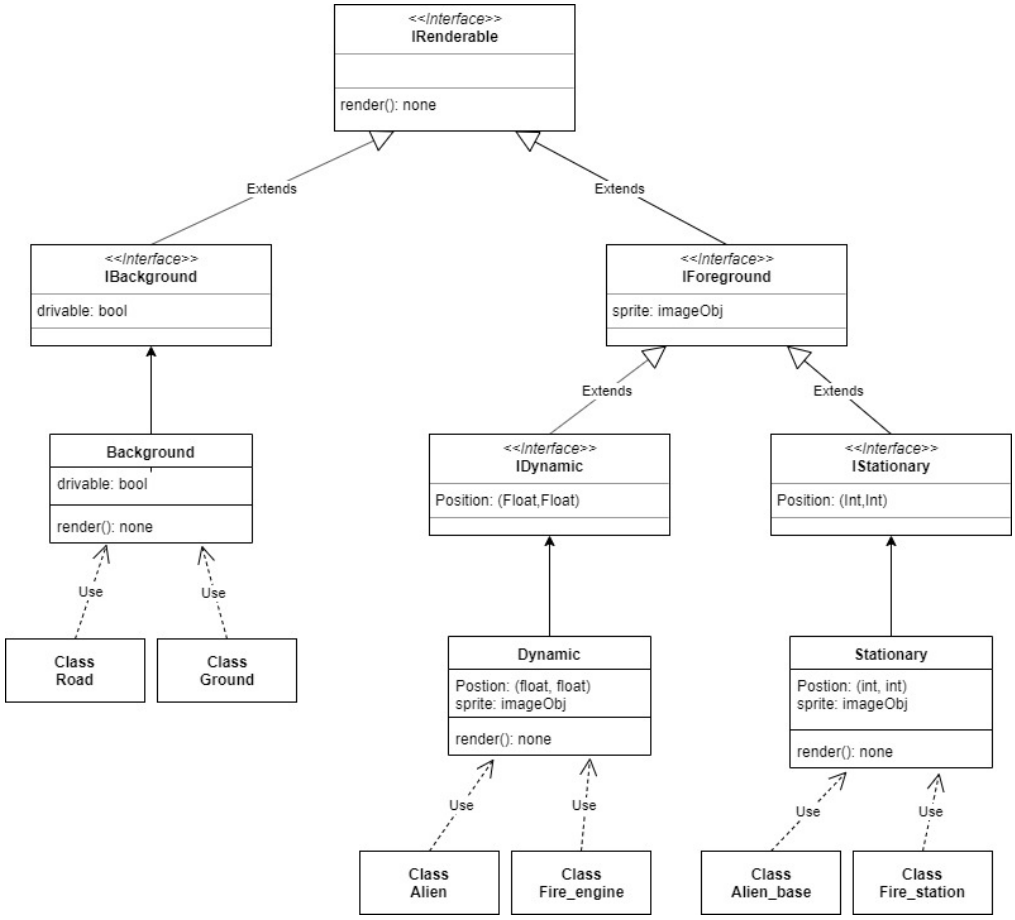
This UML class diagram shows all the main game logic classes that will be used as well as their methods and attributes. They have been separated into objects that will and won't be rendered.



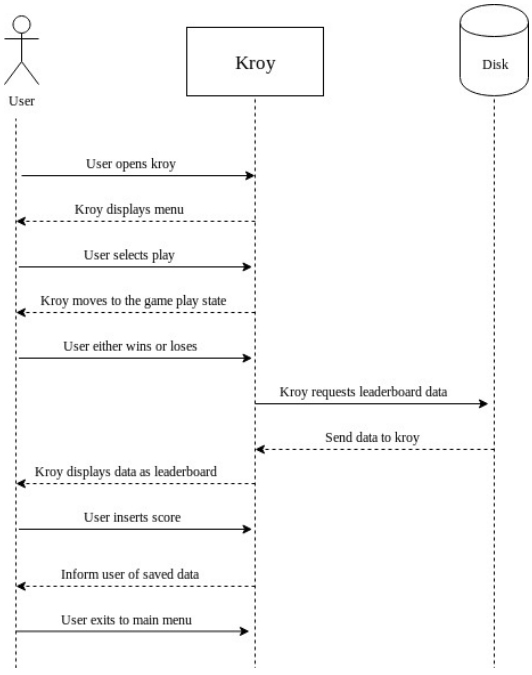
State Diagram (Figure 2)

This diagram was used to give an overall flow of the game.It shows a series of states that the system can be in. Each of these states are represented as circles which all have arrows indicating a direction that the game can go in. Initially the system will start in the "Menu Screen" state which is indicated by the arrow labelled "Start".

Rendering inheritance diagram (Figure 3)



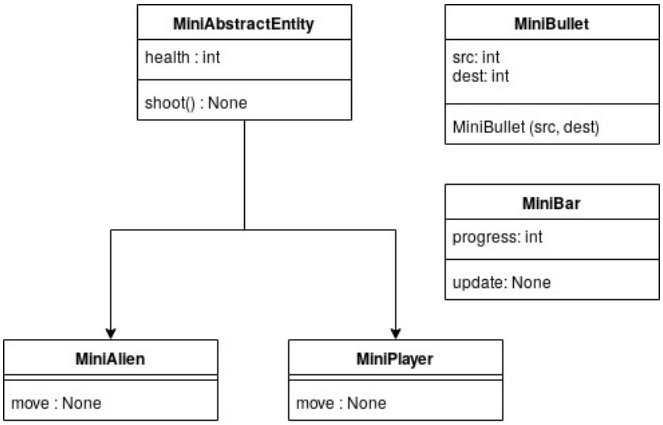
This UML rendering inheritance diagram shows how the rendering interfaces will be split up for each object within the game. It shows the hierarchy that will be implemented and which methods are intrinsic to each type of rendered object.



Sequence Diagram (Figure 4)

This sequence diagram lays out a standard interaction that the user will have with the system. In this diagram there is a single actor who will be a student or parent that will be interacting with out software. Next is the lifeline called “Kroy” which represents our system and finally we have the “Disk” which is the storage used by the environment. All of the messages sent by the actor are synchronous, meaning that the user needs to wait for a response before moving onto the next stage. We made a conscious decision to do this to avoid the complexity of having to introduce asynchronous behaviour through additional classes. This fits in with the KISS principle that was introduced during the lectures and the brief.

Minigame design (Figure 5)



This UML diagram shows all the core classes for the minigame. In this diagram the inheritance between the two entities (MiniAlien and MiniPlayer) to the parent entity is shown as well as the two other important classes, MiniBullet and MiniBar.

While considering on how to design these diagrams we came across three tools that we could use. The first one we considered was “PlantUML”. This was a feature rich system that you can download and install that contained many of the UML objects that we would need. The problem with this was that due to it being feature rich it has a steeper learning curve compared to some of the alternatives. One such alternative was google drawing which is a simple drawing system that integrates well with google drive, which we are using for storage. However this lacked the UML features that “PlantUML” had. In the end we decided to use “draw.io” which has the UML features and has an easy and convenient way to export diagrams to google drive.

3B

Figure 1

Here the classes are separated into the objects that will be rendered in the game and the objects that won’t. This separation will help the developers to know which parts need to be defined as part of the rendering hierarchy. Within the rendered objects are the fire engines and stations, the alien bases and aliens. This architecture accounts for the fact that the product requires certain attributes to be unique to each engine, for example the range. Many other important requirements from the brief have been added to this architecture an example being the destroy method on the fire station class. This will be run when the game requires the station to be destroyed as part of the scripted sequences required by the brief. Another important part of this proposed architecture is the attributes contained in the alien base class. Here it is of note that there is an increaseDefence method. This method was proposed to meet the requirement for the bases to be harder to defeat as the game continues. In an implementation, this method would be called multiple times throughout the game.
Here **(ENUM)** is used to represent a structure that only stores data which is a non standard notation just used so that the developers could see where important parts of the architecture are referencing, in this case the alien class.

Figure 2

Each state represents a screen that will be given to the player. Writing out these states in this diagram we can ensure that there are not too many states as this will make the system confusing or jarring to use. Additionally this architecture allows us to compartmentalise each of the different states of the game so we don't load anything that we don't need and can make use of encapsulation. Furthermore it clearly outlines how to reach certain states and how it should not be possible to reach a state like the "Game over screen" from the "Menu screen. We used a directed graph for this diagram as only one of the connections needed to bidirectional and the more restrictions we can apply to each state the less chance of unexpected behaviour to occur. Therefore by restricting our connections to be bidirectional we decrease our chances of bugs occurring. Finally the format of this state diagram does not follow the standard layout defined in UML. Our reasoning for this is that this format resembles that of a finite state diagram. The entire team is used to working with finite state diagrams which clarifies the purpose of this diagram. In addition, this formatting allows a good deal of abstraction which is very important in this current stage of development.

Figure 3

This inheritance diagram shows how the objects should be structured to break up the rendering process and separate it from the main game objects. This architecture choice has been made so that the game objects are not cluttered by the logic used for rendering. These rendering operations are not fundamental to how the game logic runs and therefore should be separate from the objects themselves. The interface inheritance also allows for different stages of the rendering process to be conveyed more clearly. For example this hierarchy demonstrates how the objects that move will require different operations from those that are static. This choice will allow us to find the code for certain object operations more quickly and it allows for more specific implementation details to be added to only the relevant objects later in the development process.
In this diagram 'extends' arrows have been used to show where interfaces inherit from. The black arrow shows that a class implements an interface, and the 'use' arrow shows when a class inherits from another class.

Figure 4

As explained previously, this diagram only illustrates a typical interaction between the user and the system. Due to this, the number of messages are limited so that playing the game is engaging, easy and intuitive. By drawing out this diagram we can get an impression of how the user will interact with the disk as part of the environment. The disk lifeline in the diagram is a different shape to the "Kroy" lifeline to the left. Our thought process behind this is that the disk is a different object in nature to the others and for this we used the standard UML storage symbol.

Figure 5

This diagram shows that the alien and player class inherit from the abstractEntity class. This was done to show developers that they have very similar function within the game. Both sharing health and the ability to shoot at one another. The minigame has been designed so that the players and aliens shoot. To implement this we have designed a bullet class which will be instantiated when the entity fires. Having the bullet be a class will allow it to be rendered easily because we will have experience rendering other classes for the main system. Finally a bar class has been designed this allows the rendering for this unique object to be created and separated from the rest of the code. Here the arrows represent inheritance where the head of the arrow points to the child and the tail to the parent.