StarSpeed

Project advisor:

Jens Classen, classen@ruc.dk

Group members:

Jia Hao Liang, jial@ruc.dk, 74204

Thomas Axel Paszkowski Martyna, tapm@ruc.dk, 74598

Tobias Brunton Torp Østergaard, tbto@ruc.dk, 74155

Abstract

The purpose of this report is to document and organise the development of a 2D video game with available free tools, such as Unity for the game engine, Visual Studio Code for programming in C#, GitHub for collaboration, Midjourney for visual illustrations, and Kanban Board for work management. The developed game draws inspiration from Space Invaders. Space Invaders is a fixed shooter game from an early era of arcade video games, it was released back in 1978. The game we've developed has the enemy mobs movement behaviour based on Craig W. Reynold's boid algorithm model. The boid algorithm is based on a model visualising the synchronised movement of a flock of birds, school of fish, or a herd of land animals. The model presented 3 rules, which describes the social interactions with each other, based on their current position and velocity of other boids. The basic rules are **Separation** to avoid crowding, **Alignment** to move in a similar direction, and **Cohesion** to maintain a group formation. The algorithm enforces the enemies to be engaging compared to the simplistic movement of the aliens in Space Invaders.

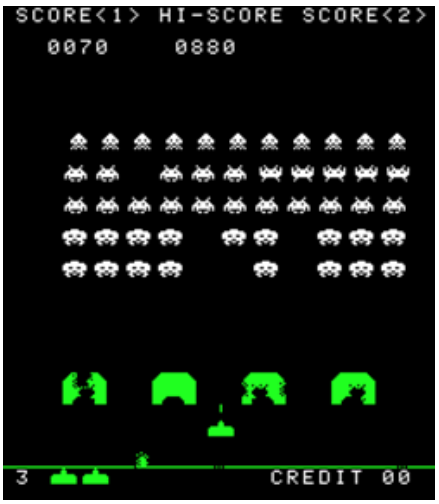
Table of Contents

1. Introduction

There are multiple variables and aspects when it comes to developing a game. A game is a form of entertainment, therefore the main objective is the users enjoying and having fun with the game. Whether a game is good looking or not, the game would not be enjoyable if the game mechanics are boring[1]

Game mechanics are the heart of the game, which sets the rules, processes and data of a game. They define the interactions between players and the game, how the play progresses in the game, the timing of events or actions, and which conditions determine the victory, defeat, life, or death of a player. Game mechanics creates gameplay, understanding, and designing the core game mechanics is detrimental to whether the game will be fun or boring[2]

The game we've developed draws inspiration from an old arcade game called Space Invaders (1978). Space Invaders has a very simple concept. The player controls a spaceship located at the bottom of the screen. The movement of the ship is locked horizontally (left/right) and the main objective is to survive as long as possible while shooting down the enemies – the aliens. By killing the aliens, the player earns points that will keep the highest records on a scoreboard. The aliens spawn from the top and move horizontally and each time they move down a row, they'll have their movement speed increased. The closer to the player the faster they will be. That's not all, the aliens have the ability to shoot bullets, which forces the player to either dodge the bullets or hide behind the 4 green walls[3]



[1](Adams & Dormans, 2012, xi)
[2] (Adams & Dormans, 2012, 1)
[3] (Anderson et al., 2015, 2)

Figure 1: Space Invaders Game (Roeder & Wolanski, n.d.)

To achieve an enjoyable game for the players we will need engaging core mechanics, which will dictate the gameplay. By implementing Craig W. Reynold's boid behaviour model into the enemy objects is a step into the right direction in creating a game with exciting game style.

1.1 Enemies Optimisation

We aim to design the enemies' behaviour to rely on the player's movements. This is because having enemies move without considering the player would be too simplistic, akin to 'Space Invaders,' where the enemies are not affected by the player and follow predetermined patterns. To meet our game's design goals, we incorporated the 'boids algorithm.' This algorithm is ideal because it enables complex, coordinated movement among groups of enemies, creating a more immersive and challenging experience. It allows the enemies to dynamically respond to the player's actions, ensuring each game session is unique and engaging. The boids algorithm simulates natural flocking behaviours, commonly observed in birds and fish. It operates on three fundamental rules: separation, to avoid crowding; alignment, to move in a similar direction; and cohesion, to maintain a group formation. These rules enable complex,coordinated movements, making the algorithm ideal for controlling groups of enemies in a game environment. Below we will explain how the boid algorithm works:

Separation:

Each boid avoids running into each other, so if two boids are too close to each other (interfering with each other's protected range) they will steer away from each other.
Here is how it happens:

1. At the start of the update for the boid, we have two accumulating variables *(close_dx and close_dy)* which are zero.
2. We loop through every other boid. If the distance to a particular boid is less than the protected range, then:

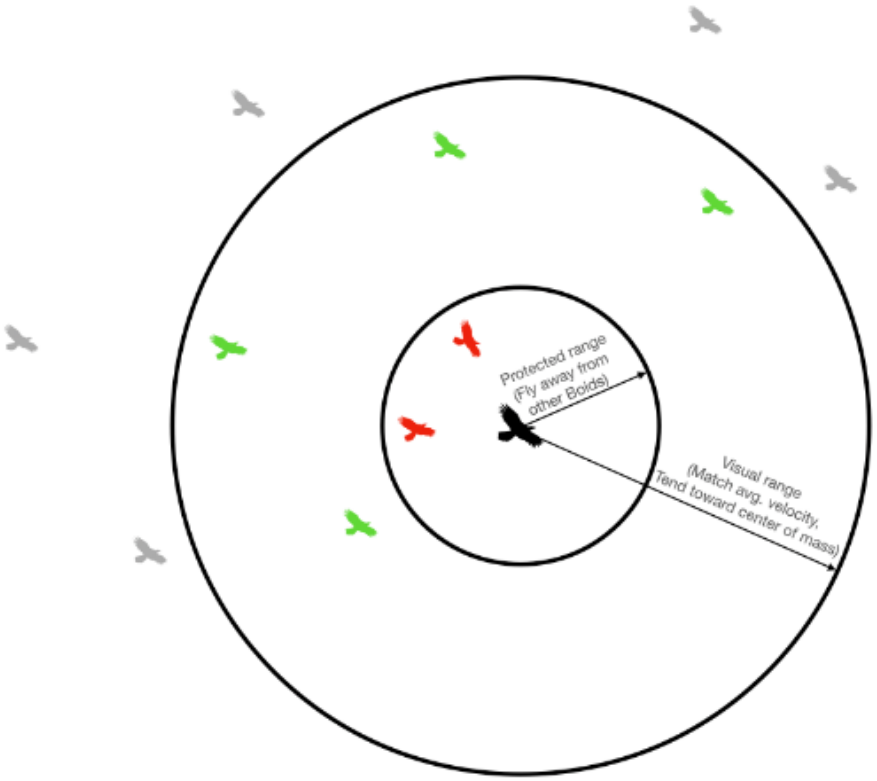
$$close_{dx} \mathrel{+}= boid.x - otherboid.x$$
$$close_{dy} \mathrel{+}= boid.y - otherboid.y$$

3. Once we've looped through all other boids, then we update the velocity according to:

$$boid.vx \mathrel{+}= close_{dx} * avoidfactor$$

$$boid.vy \mathrel{+}= close_{dy} * avoidfactor$$

(avoid factor is a tunable parameter)[4]



Each boid determines whether each other boid is in its protected/visual range.

Figure 2: Separation[5]

Alignment:

Each boid attempts to match the velocity of other boids inside its visible range. It does so in the following way:

1. At the start of the update for a particular boid, three variables (xvel_avg,yvel_avg, and neighbouring_boids) are zeroed
2. We loop through every other boid. If the distance to a particular boid is less than the visible range, then:

[4] ("Boids-algorithm")
[5] ("Boids-algorithm")

$$xvel_{avg} \mathrel{+}= otherboids.vx$$

$$yvel_{avg} \mathrel{+}= otherboids.vy$$

$$neighboring_{boids} \mathrel{+}= 1$$

3. Once we've looped through all other boids, we do the following **'if neighboring_boids>0'**

$$xvel_{avg} = xvel_{avg}/neighboring_{boids}$$

$$yvel_{avg} = yvel_{avg}/neighboring_{boids}$$

4. We then update the velocity according to:

$$boid.vx \mathrel{+}= (xvel_{avg} - boid.vx) * matchingfactor$$

$$boid.vy \mathrel{+}= (yvel_{avg} - boid.vy) * matchingfactor$$

(where matchingfactor is a tunable factor)[6]

[6] ("Boids-algorithm")

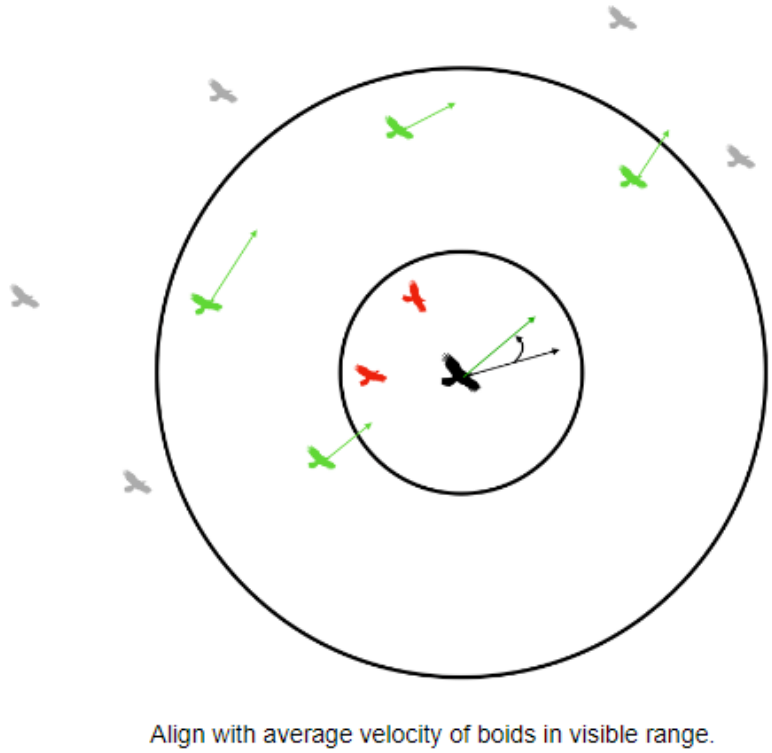


Figure 3: Alignment[7]

Cohesion

Each boid steers toward the centre of mass of other boids within its visible range. It does so in the following way:

At the start of the update for a particular boid, three variables ($xpos_{avg}$, $ypos_{avg}$, *and* $neighboring_{boids}$) *are zeroed*

We loop through every other boid. If the distance to a particular boid is less than the visible range, then:

$$xpos_{avg} \mathrel{+}= otherboid.x$$

$$ypos_{avg} \mathrel{+}= otherboid.y$$

$$neighboring_{boids} \mathrel{+}= 1$$

[7] ("Boids-algorithm")

Once we've looped through all other boids, we do the following **if neighboring_boids>0:**

$$xpos_{avg} \mathrel{+}= xpos_{avg}/neighboring_{boids}$$

$$ypos_{avg} \mathrel{+}= ypos_{avg}/neighboring_{boids}$$

We then update the velocity according to:

$$boid.vx \mathrel{+}= (xpos_{avg} - boid.x) * centeringfactor$$
$$boid.vy \mathrel{+}= (ypos_{avg} - boid.y) * centeringfactor$$

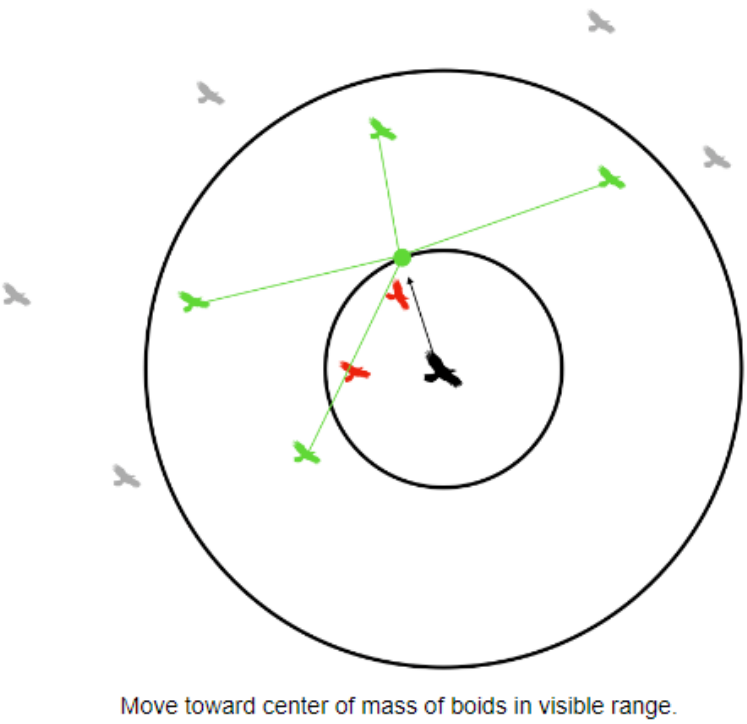


Figure 4: Cohesion[8]

[8] ("Boids-algorithm")

1.2 Outline

The project is divided in 8 sections, it begins with section [2] that describes which tools we have used to solve this project, such as Midjourney for designing some of the game objects, and Unity, as a specific tool used when creating games. Section [3] game design, covers how the game was designed, in unity and how we added our own design from mid journey to Unity, we also explain the behaviour and the relation between the ‘enemy boids’ and the ‘player/spaceship’, wich different types of enemies there are Section[4] implementation covers our code, here we will explain the code and the implementation. Section [5] User manual covers how we control the spaceship, and how we shoot. Section [6] Testing covers the testing of the behaviour between the enemies and the player, if they behave as intended, does it work as we designed it to. [7] discussion, here we will discuss the differences between our intentions with the game, and how the final result looked when we were done, and what we could have done to strengthen the user experience, or in general make it better. [8] conclusion here we will address what we ended up with, and if the final product works as intended.

1.3 Problem Definition

In this project, we aim to explore the advantages of using modern game engines like Unity by applying them to classic games. Our focus will be on how these advanced tools can enhance an old school game like ‘space invader’. We intend to reimagine ‘Space Invader’ as it might appear today, taking advantage of the most recent advancements in game development. This will help us understand the influence of contemporary game design techniques on traditional gaming experiences, and experience the process of developing a game.

2. Tools

This section describes the tools we have been using to make the game and how we delegated and organised the workload among ourselves. We have been using tools, such as Unity as the game engine, Visual Studio Code as our IDE (Integrated development environment), GitHub, Midjourney, and Kanban Board.

2.1 Organisation

To coordinate our project, we used meetings to form the idea about how the game should work, we started by outlining on paper, and moved the process to the whiteboard, for detailing the structure and the movement of the game. We have used GitHub to share the code between each other, it made it alot easier for us to work individually, while still being able to share the changes made in unity. We have used discord for our meetings and for coding and screen sharing while handling specific issues or for showing detailed explanations regarding the code. We have used a kanban board to organize the different tasks and to assign tasks within the group. We have used a class diagram to visually represent the structure of our game. It helped us understand and define the relationships between different classes, including their attributes and methods. By mapping out these classes, we could better organise the games functionality and ensure communication and the collaboration within the group. The diagram is our blueprint, guiding us in implementing the game's mechanics, and interactions.

2.2 Unity

Unity3D often simply refers to Unity as a game engine and IDE for developing typically video games. Unity is well known for its ability to fast prototyping and publishing for an extensive amount of targets. Unity's toolset to create a video game consists of executing the task of graphics, audio, physics, interactions, and networking

Unity has the ability to use a variety of platforms using the exact same code and assets. Unity supports 4 main categories, which are Mobile, Desktop, Web, and Consoles. Unity can build iOS, Android, BlackBerry and Windows Phone 8 for mobile. While for desktops, Unity can create executables for Windows, Windows Store, Mac OS X, and Linux. Unity supports these browser based options, such as Unity Web Player and Google Native Client.

2.2.1 Unity Editor

The editor in Unity is made out of several sub-windows and most commonly used are the Project Browser, Inspector, Game View, Scene View and Hierarchy.[9] The Project Browser is

[9] (Haas, 2014, 12)

where you find all the available and imported assets in Unity. Importing assets in Unity is simple, the user must drag it into the Project window, and then Unity will do the rest of the job. Unity will automatically update the assets with their latest mods across the project.

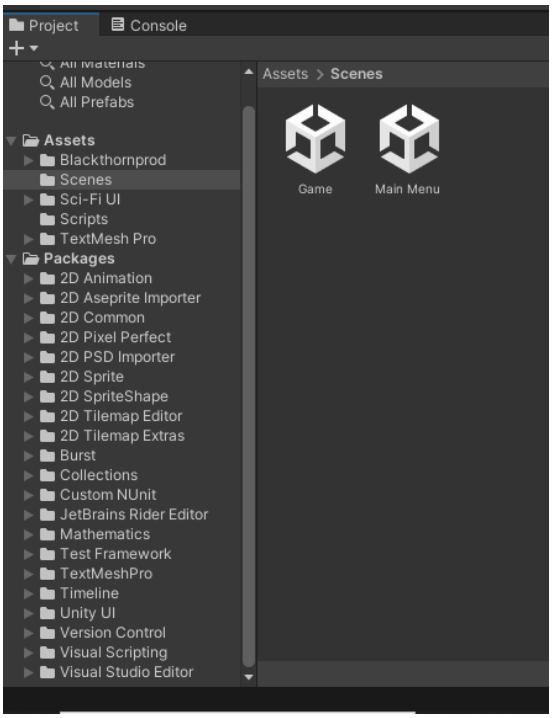


Figure 5: Project Window

The Inspector window is where all the GameObjects (GO) are found, and the values of GO can be adjusted. The Inspector window also shows which components are tied to the GO like Scripts, physics, Colliders, and sounds. The Inspector window also allows the users to either assign or change the values on the GOs tied to the scripts.

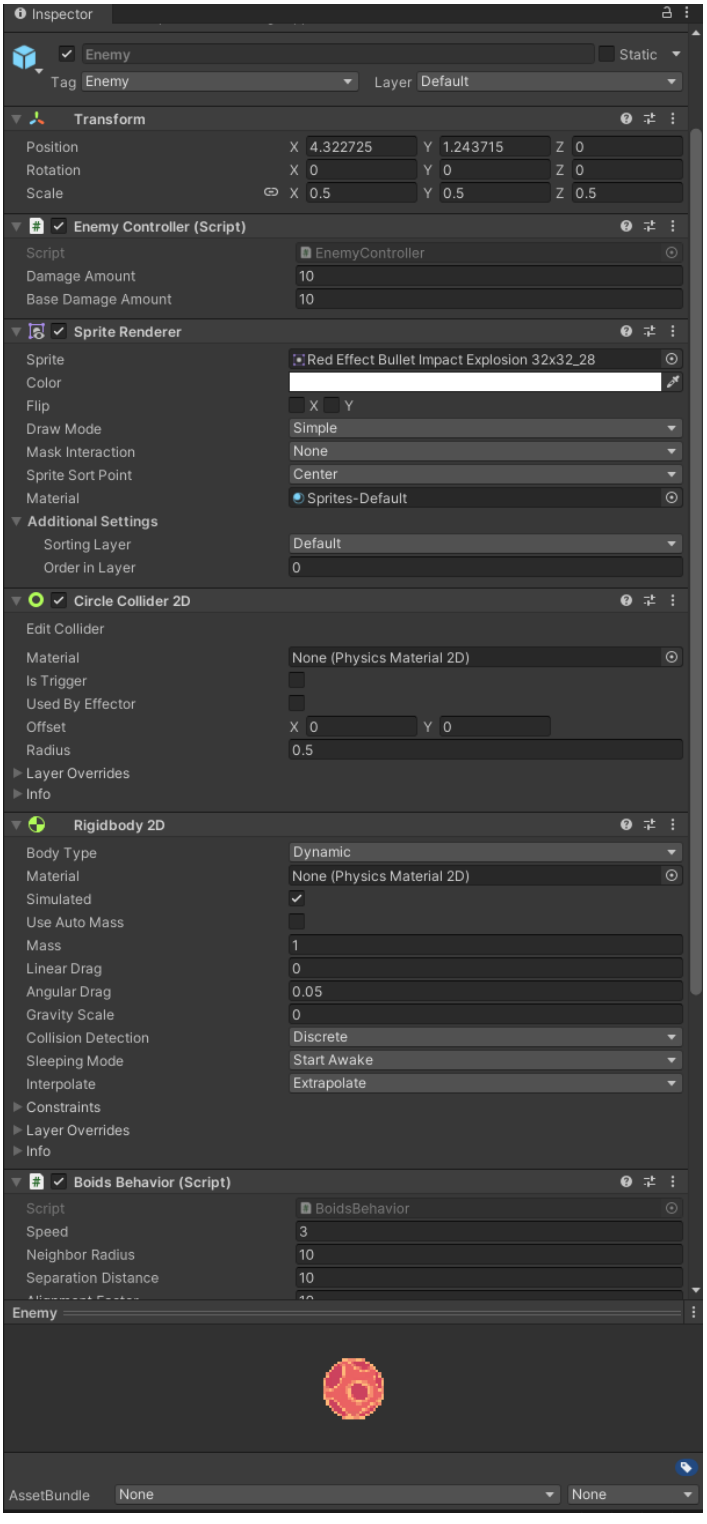


Figure 6: Inspector Window

The Game View allows the users to get a preview of the game and what you see is what you get in the end. This window permits the developers to input their changes without having to

wait for the project to be compiled and deployed. The Scene View is the construction site for the game, where the developers can drag and drop their assets from the Project Window. In the Scene View there is a 3D control handle with grid snapping for placing the objects into the desired position.

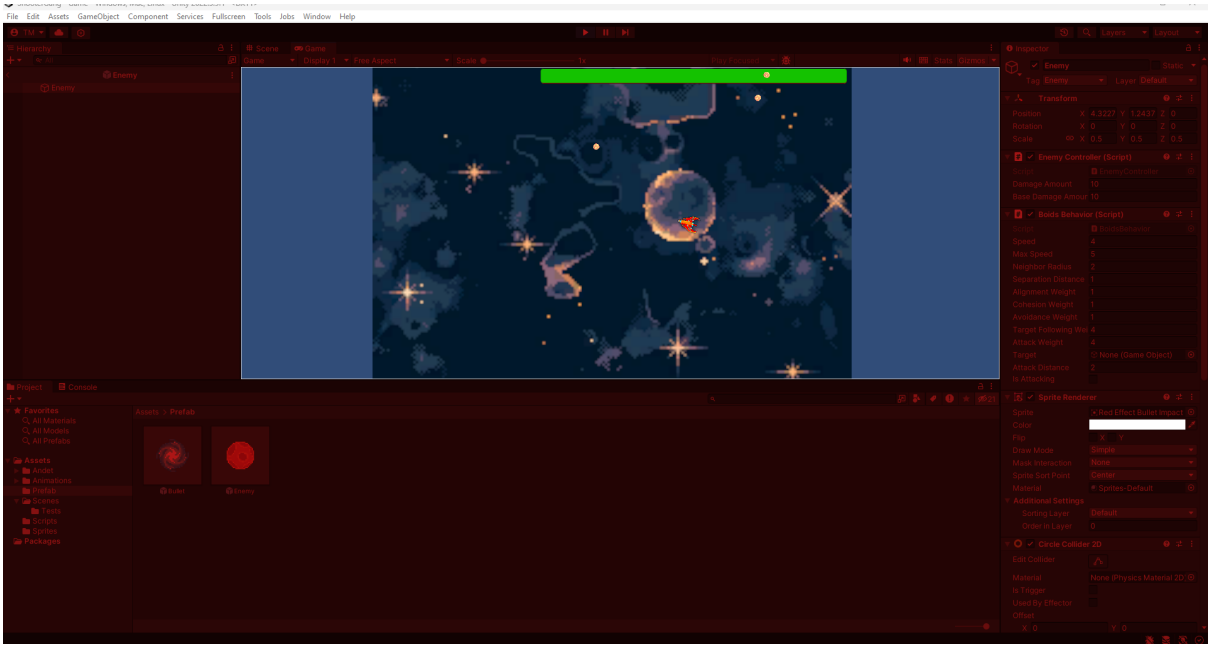


Figure 7: The Game View

The Hierarchy window is made up of a list with all the objects that are currently in the scene. The list is updated automatically whenever an object is added into the scene. Another feature is that the developers can assign a parent or child relationship for objects by dragging them on top of each other.[10]

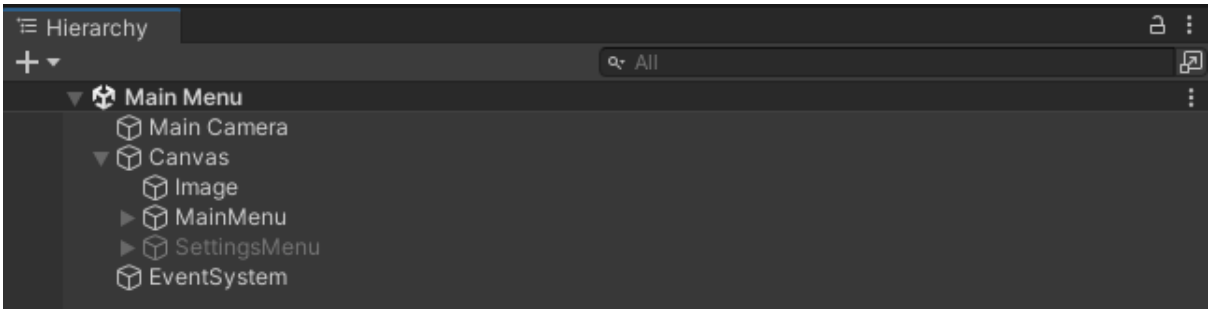


Figure 8: Hierarchy Window

[10] (Haas, 2014, 12-15)

2.2.2 Scripting Languages

Unity provides 3 ways to build a program in their IDE. The 3 programming languages are Unity Script, C#, and Boo. These languages run on Mono, which has a fast compile time and is an open-sourced version based on Microsoft's .NET platform. In addition to the above mentioned abilities the platform also supports cross compatibility on other platforms . As for the 3 scripts both UnityScript and C# is widely more popular and used compared to Boo. Boo shares similarities to the syntax of Python but at the same time its structure is similar to UnityScript. It's problematic to troubleshoot the code, due to few people using Boo and therefore makes it harder to find anyone else that can assist you. UnityScript has more users and makes it easier to get assistance from either searching and finding other people's codes or asking for help on the Unity forum. UnityScript is similar to Javascript and makes it a better option for beginner and novice Unity users, due to it being easy to learn and fast typing. It handles the type casting behind the scenes and the user can freely switch between the typing style to be either dynamic or strict. Another feature UnityScript has is the ability to make classes with inheritance. The last scripting language in Unity – C# is slightly more challenging to learn and master. C# gives the user a more complete and precise control over the program being coded. C# require the users to manage the components manually because unlike UnityScript, C# does miniscule behind the scenes. For our project we chose to use C# to write the code for the program. As C# offers a slight more control over the behind the scenes.[11]

2.3 GitHub

GitHub is a platform that supports development of software programs and applications. GitHub functions as an online social network for programmers. The users of GitHub can create their own repositories and work with other programmers that have registered as a user on GitHub. The users can furthermore join other users' projects and contribute to them.[12]

[11] (Haas, 2014, 19-21)
[12] (Jarczyk et al., 2014, 80-81)

GitHub uses Git as its revision control and source code management system, allowing earlier versions of the projects to be recovered at any time. The programmers can review the project's timeline and history to see which, who, when and why the changes were made.[13]

A repository is a projects library that stores the entire collections of files and folders associated with a project along with an archive for each of the file's revisions. The file history is shown as snapshots in time called "commits ". The commits can be grouped into various lines of development, also termed as "branches".

Fork is a term used when a new repository is created with identical code or similar visibility settings with another repository that is already existing. Forking is usually used for iteration on ideas or changes on open-source projects, before they are proposed back to the original repository, or when a user does not have the access or any rights to write in the original repository[14]

2.3.1 Git commands

The usage of Git consists of specific commands that allows the developers to copy, create, change, and combine codes. The commands can be executed from the command line or inside the GitHub Desktop application. Some of the most used commands are[15]:

- **git init** – Create a new repository and start tracking an existing directory. It also adds a hidden subfolder that holds the internal data structure for version control.
- **git clone** – Create a local copy of a project. The copied clone has everything the original has like the project's files, history, and branches.
- **git add** – Performs staging, which is the first part of the two steps process. The changes become a part of the next snapshot and the project's history.

[13] (*About Git*, n.d.)
[14] (*Fork a Repository*, n.d.)
[15] (*About Git*, n.d.)

- **git commit** – Save the snapshot to the project history and complete the process of tracking the changes. Anything staged with **git add** will become part of the snapshot with **git commit**.
- **git status** – Display status over changes as untracked, modified, or staged.
- **git branch** – Display the branches being locally worked on.
- **git merge** – Combine the changes of two branches together. Usually merging a feature and a main branch.
- **git pull** – Update the current working branch, and all the remote tracking branches.
- **git push** – Update the local commits to the corresponding remote branches. You can think of it as an update or publish command line.

We've primarily used Github as a tool to upload and share codes between the group members. As it can quickly become a mess if we just send a file instead, due to the lack of visibility of the updates made on the code files. We have made an organisation and created a private repository and used some of the above-mentioned Git commands like **git init**, **git pull**, **git push**, etc. to keep track of the timeline of the project history, update changes, and retrieve the recently updated file.

A problem that we have encountered with the use of GitHub is when 2 users push at the same time, which results in the program not being able to push the code into the repository.

Underneath is a picture showing the layout of the github desktop application, and how you would push the code into the repository.

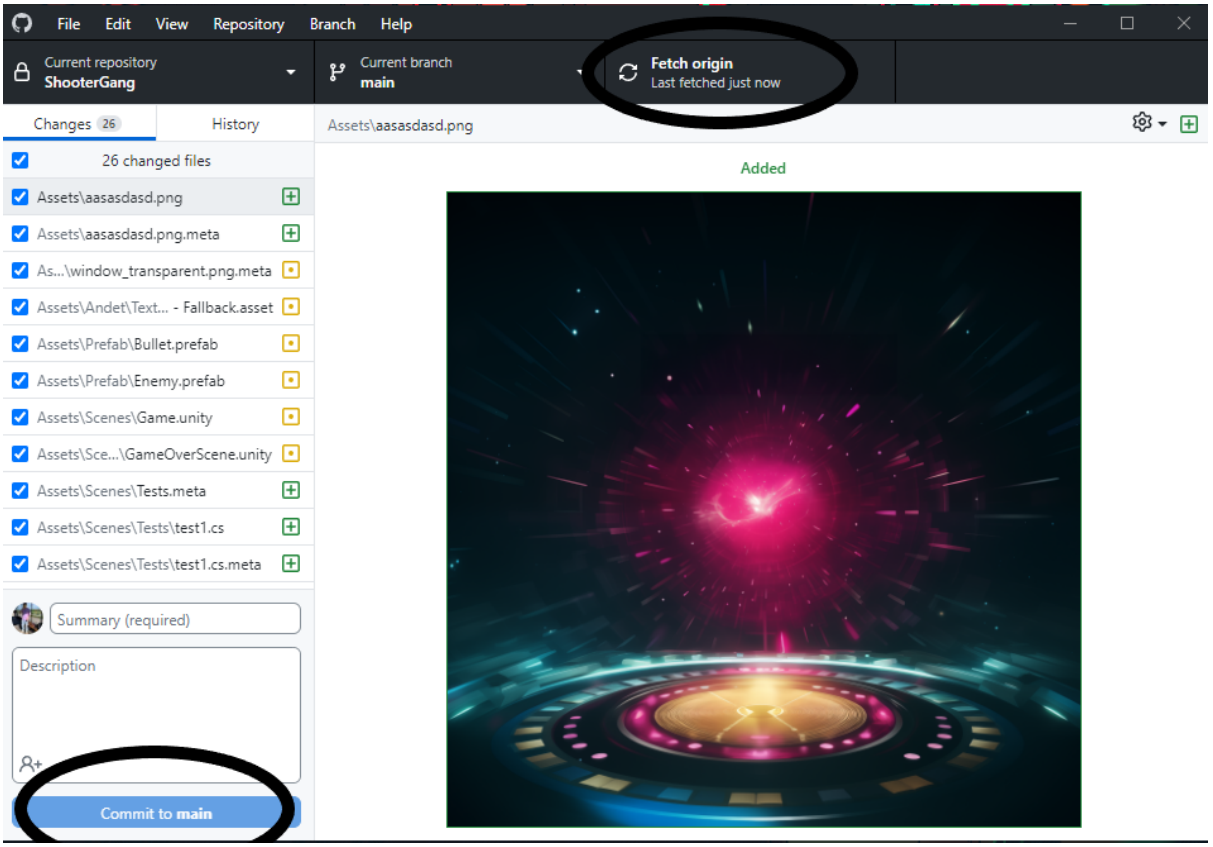


Figure 9: GitHub Layout

2.4 Visual Studio Code

Visual Studio Code is an IDE tool used to write, test, build, package, and deploy various software programs. It holds a feature that gives Git control abilities. Visual Studio Code has the option to work and export text files in multiple programming languages, such as C#, C++, Clojure, F#, HTML, JSON, Java, Lua, PHP, Perl, Python, SQL, Visual Basic, XML, and etc. Furthermore Visual Studio Code supports development in Node.js and ASP.NET[16].

We decided to use Visual Studio Code as the IDE for the project and the reasoning behind it is that it supports the features for writing in C#, compiling, and debugging in the editor. Although the main reasoning is that scripting in Unity is done in C#, which is also an object oriented programming language.

[16] (Visual Studio Code, 2019, 1-2).

2.5 Midjourney

The field of Artificial Intelligence (AI) has progressed with major technological advances and keeps improving day by day. The use of AI has slowly been seen in certain areas. One of those being the field of creativity – Art. Art is diverse and is appreciated in different ways by the individual. With the recent expansion of AI-Art, such as Midjourney and has made expressing creativity through digital art effortlessly. Midjourney is only available on the application called Discord. The way Midjourney operates is by doing a "prompt", prompts are based on a text to illustration in Midjourney and a system to check whether the AI registers the prompt as one[17].

We used Midjourney as a means to an end due to having limited skills and proficiency in arts and computer graphics, we then touched minor errors up in Photoshop, also using their AI. The decision was made since the main focus is to have a functional game and we tried allocating the time spent to code the program

Going through the way we used Midjourney in our project. We started by writing this specific prompt "/Imagine prompt" and then our request. In this case it was /imagine prompt game over screen for a game called starspeed, we to be a little less specific to see what kind of result we would get:

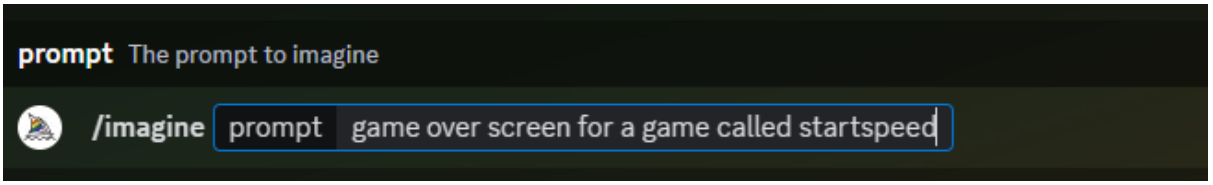


Figure 10: An example of a prompt

[17] (Radhakrishnan, 2023, 94-96)

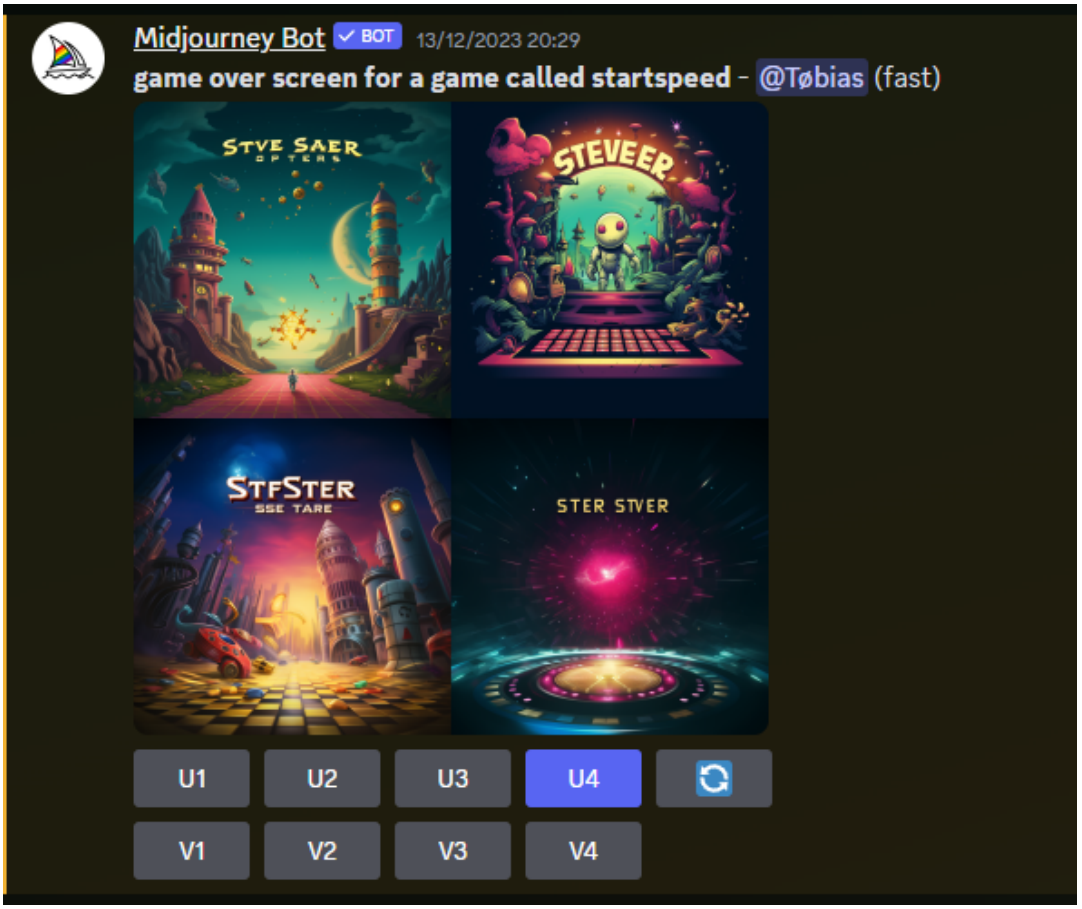


Figure 11: The 4 results of the prompt

Under the four versions the AI created there are some buttons, these are represented by U and V followed by numbers also a redo button. The U is a button which upscales the picture, and creates a more detailed, and bigger sized files of the picture and the V is a button which creates 4 new variants taking inspiration the one of the four pictures you chose, the numbers all represent which picture of the four you wanna either upscale or create variants of
In this case we liked number 4 so we chose U4 which means upscale number 4.

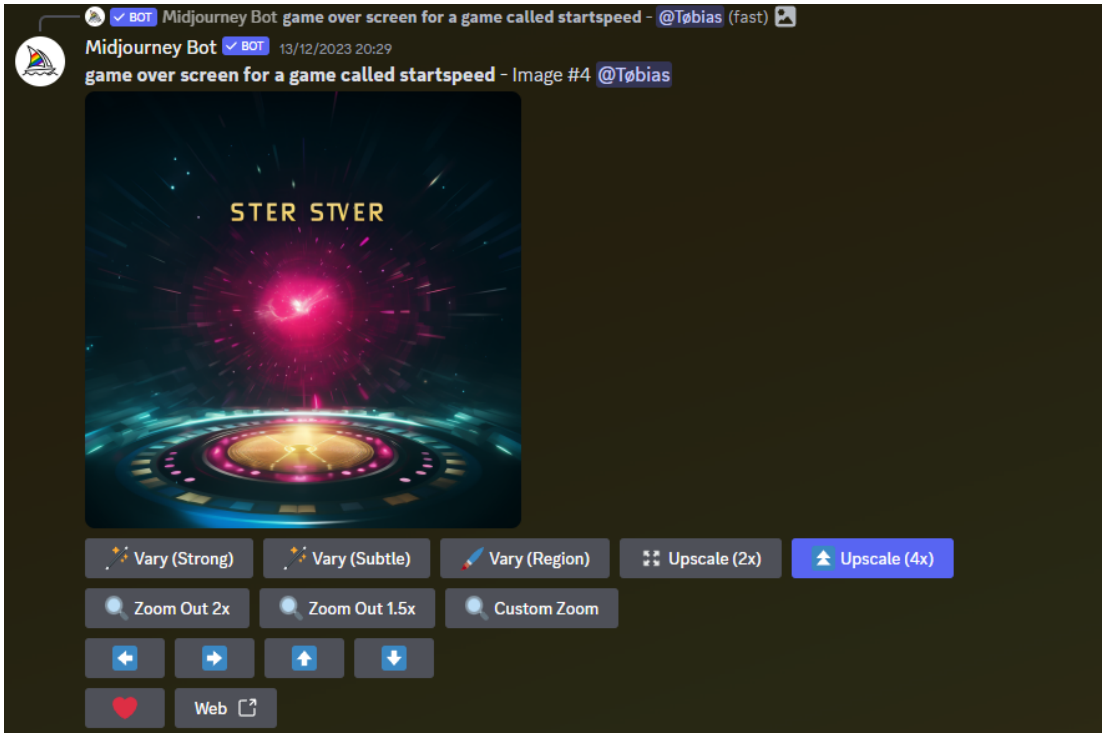


Figure 12: Upscaling and variations

After choosing to upscale a picture you get a lot more options, where you could vary it in different ways, upscale it, zoom it out or in and move the position of the picture. In this case we just upscaled it 4 times more to get a good size image to use in our game.

Midjourney is still not good at handling text, so we then chose to move over to Photoshop to remove it. So it was usable in the game.

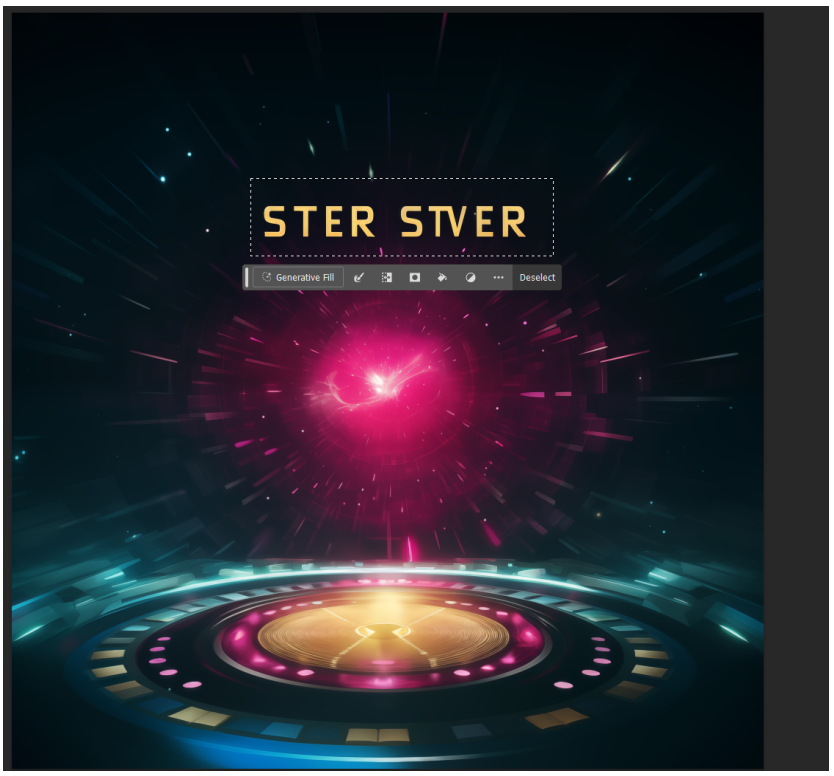


Figure 13: Adobe Photoshop

In Adobe Photoshop we used once again an AI, we marked the area, pressed generative fill and left the text box empty so that it would remove and blend it with the area outside the marked area. This is the result that we used for the games background:

Figure 14: The final result

2.6 Kanban board

Kanban board uses Card, Column, Swimlanes, and WIP, so that teams can visualise and manage their workflows effectively[18]. Each column represents a phase in the workflow. It is typically divided into three different stages: “To Do”, “In progress”, and “Done”.
Each task in the project is represented by a card that we place in the relevant column. The

[18] (“What Is a Kanban Board and How to Use It? Basics Explained.”)

card stores information about the task, the description, a person that is responsible for the task to be done, and a deadline. The Kanban board visualises the workflow, so all that are involved with the project can see the status of the tasks, and where the different tasks are in the process. The tasks are moved from left to right on the board, which reflects the workflow. One of the main reasons to use Kanban board is to limit the amount of "work in progress", so we have a limit for how many tasks that can be in the different phases (WIP), this ensures that a task is "Done" before a new task is in progress. This helps us optimise the workflow in the project.
on the model down below, you can see our Kanban board, and the tasks we have for this project.

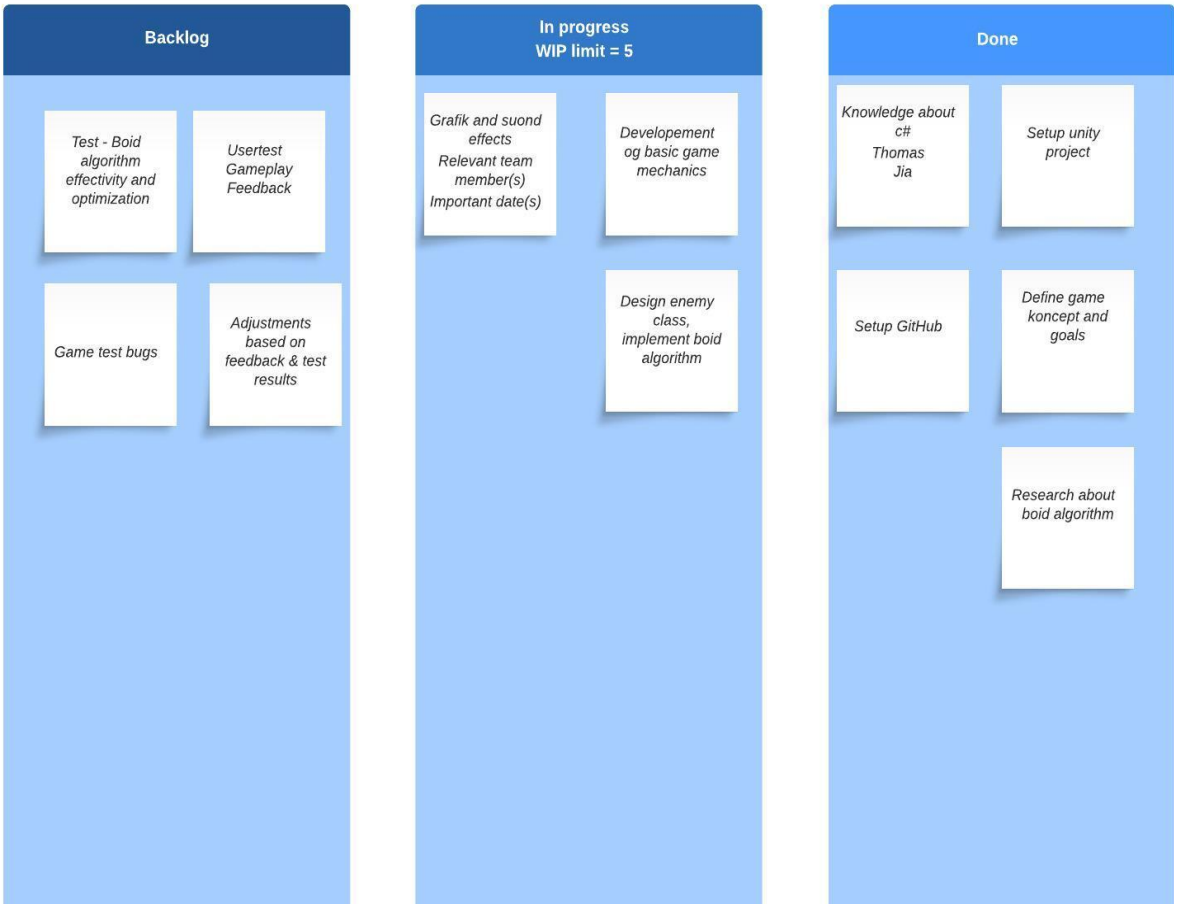


Figure 15 : Kanban Board

3. Game Design

To ensure the reader understands the game and how it works, we will explain how this game is designed, they way this game works, is that you look down on the map, you are the player controlling the ship located in the center of the map, and as the round begins the player will have to avoid enemies that are aiming for player, the player will have to shoot all the

enemies, and when all the enemies are dead, the next wave comes, where the enemies, will be a little stronger, and the parameters for the algorithm will also change, so the game will be unpredictable. Figure[5], is an early mock up of the game, and how we intended the game to look, here we can see 4 different enemies, the idea was for the enemies to have 4 different behaviours based on the algorithm, which is not very different from the result.

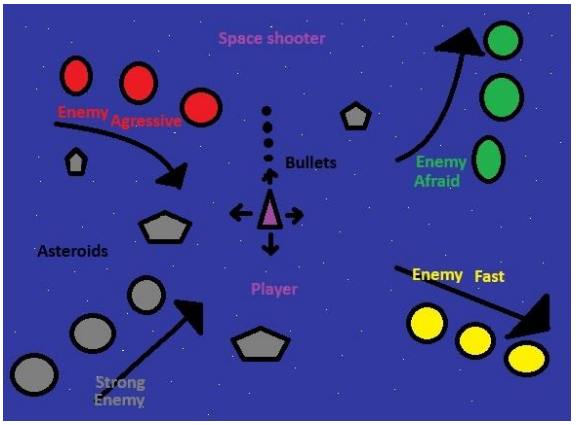


Figure 16: Mockup

3.1 Scene

There are 3 different scenes in the game which allows for the game to have different states. The scenes are as follows, Main Menu scene (0), Game scene (1) and Game over scene (2). The scenes all have different purposes and are all set up differently, the Main menu scene only focuses on the UI part as there are only buttons in there, the play buttons changes the scene from the Main menu to the game scene. The game scene are where the game part happens, where the player, enemies, bullets and more are present, and not any not necessary gameobjects, in this case when ever the players health reaches 0 or below, the scenes once again changes from being game to now game over, here it once again removes any not necessary game object to make sure only the scores and the play again, or main menu buttons are present.

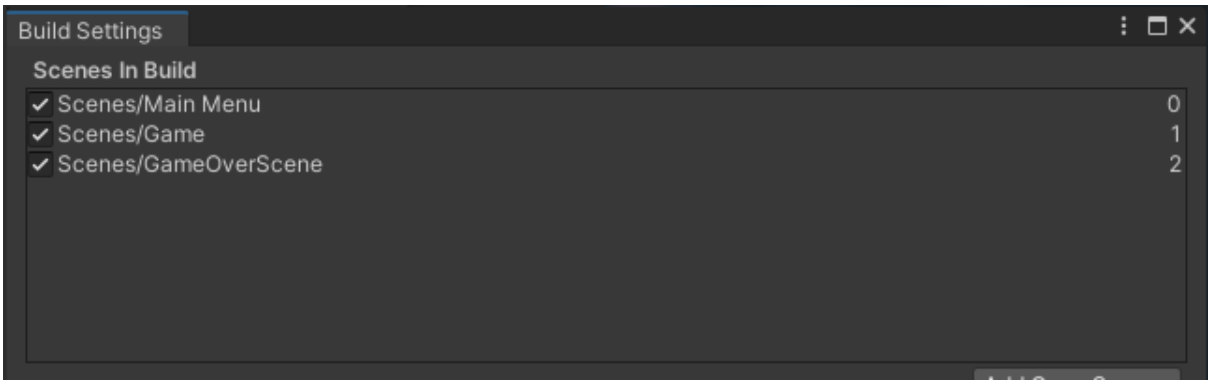
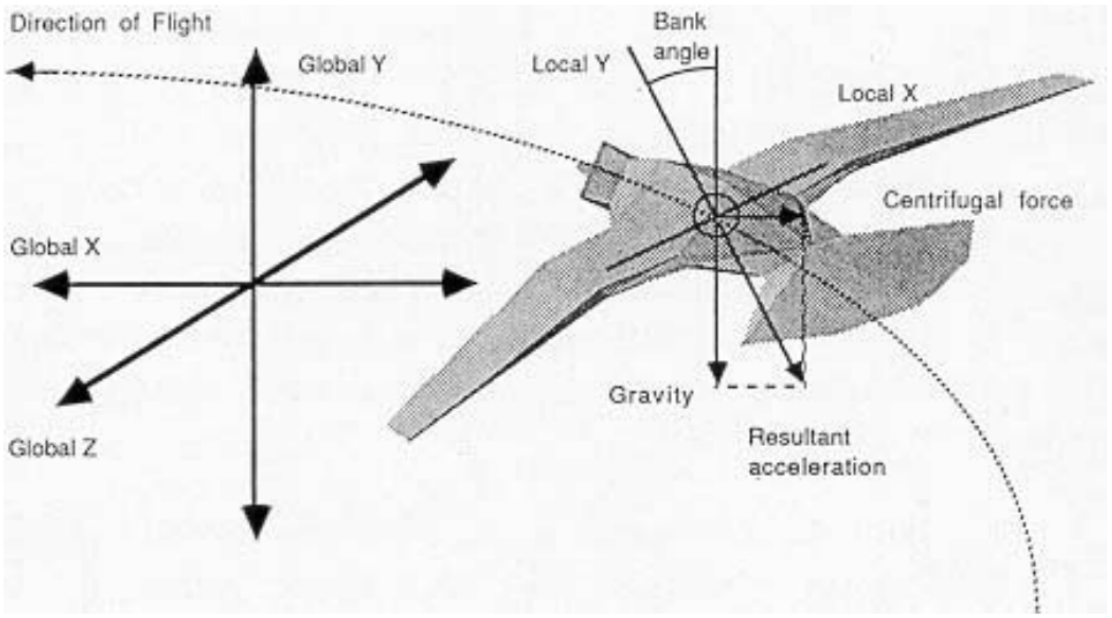


Figure 17: Scenes

3.2 Enemies

The enemy class in our game is using Reynolds' boid model, which is usually represented as a flock of birds. The behaviour of flocking consists of a balance between 2 opposing traits - A desire to stay close to the flock and a desire to avoid collisions within the flock. Building a simulation of a flock of birds requires a model that supports geometric flight. Flying geometric is when an object moves continuously in a "forward direction" (local positive Z axis) and adjusting the angle using steering rotation (local X and Y axes), which realign the object's orientation of the Z axis.The model of flight in flying geometrics relies on the object's own coordinate system to navigate around. The X axis represents the "wings" and ability to turn either left or right, the Y axis represents the "eyes" ability to look up or down, and lastly the Z axis represents the ability to move forward or back. Envision the motion of turning and moving at the same time, while staying in the motion of flying[19]

The enemies in the game differ from the model due to having it programmed as a 2D game, where we do not use the Z Axis as part of the enemy boid. However it was included in the explanation to give the general idea how the boids move according to Reynolds model.



[19] (Reynolds, 1987, 27)

Figure 18: Geometric Flight[20]

3.3 Boid algorithm /Enemies behaviour

In the early stages of Reynolds' boid model each of the 3 each of the 3 behavioural characteristics will form their own individual suggestion on the boid's movement. These suggestions are expressed as *acceleration requests,* which come in the form of a 3D vector with a limited unit magnitude. Imagine if each behaviour characteristic said, "If I were in charge, I would accelerate in that direction.". The navigation module of the boids "brain" collects, compares, and considers these requests, then determines 1 single acceleration request will take action. This process involves combining, prioritising, and resolving the conflicting behavioural characteristics. The navigation module sends the acceleration request to the flight module, which will try and fly in that direction.

One of the possibilities to combine the acceleration requests is to average them, taking the strength into account. Strength in this context is a parameter with value between 0 and 1 and is associated with each behaviour characteristic. However, this approach finds its limitation when faced in situations that require fast and critical thinking, which is necessary to avoid obstacles. Another method Reynolds used as the final example in the 1987 model was to have the implementation to prioritising acceleration allocation, where each of the 3 behavioural characteristics will be assigned a strict priority order. The navigation module will consider the priorities that are added into an accumulator, where the unit magnitude of the requests will be measured and added to another accumulator. That process will continue to happen, until the acceleration value does not exceed the maximum acceleration value. As follows the boids acceleration is controlled by the priority of the 3 behavioural characteristics. In emergencies, the most pressing need will take priority first, if all available acceleration is gone, then the less pressing behaviours may be temporarily ignored[21].

[20] (Reynolds, 1987, 27)
[21] (Reynolds, 1987, 29)

3.4 Game rules

There is no real winning condition as the game works around the scoreboard and getting enough points to land within the highest scores, which will be shown on the scoreboard.
The objective in StarSpeed is identical to Space Invaders. The player has to survive as long as possible, while shooting down the enemies and avoid collision with the boids, otherwise you'll take damage. There will be some suggestions of ideas in section 7, where we discuss different ideas that can be implemented in the future.

3.5 Player

The primary role of the player is to navigate through an endless stream of enemies and challenges with the goal of achieving a high score.
The player must manoeuvre their spaceship with WASD or the arrow keys to avoid enemies
The player can shoot enemies to destroy them, with the mouse which also allows the rotation of the player to be calculated.

4. Implementation

This section will cover the details and the process of the game's implementation. The game is developed in unity. By using unity we have had benefits with the tools provided by the game engine, because unity is a game engine that includes tools specifically used for game development. This way of coding and resigning scripts to objects and scenes, is a new way of coding, and differs a lot from java coding in general. Before reading the implementation, we suggest reading the tools section[2], which shows all the different tools we have used for this during the development of the game.

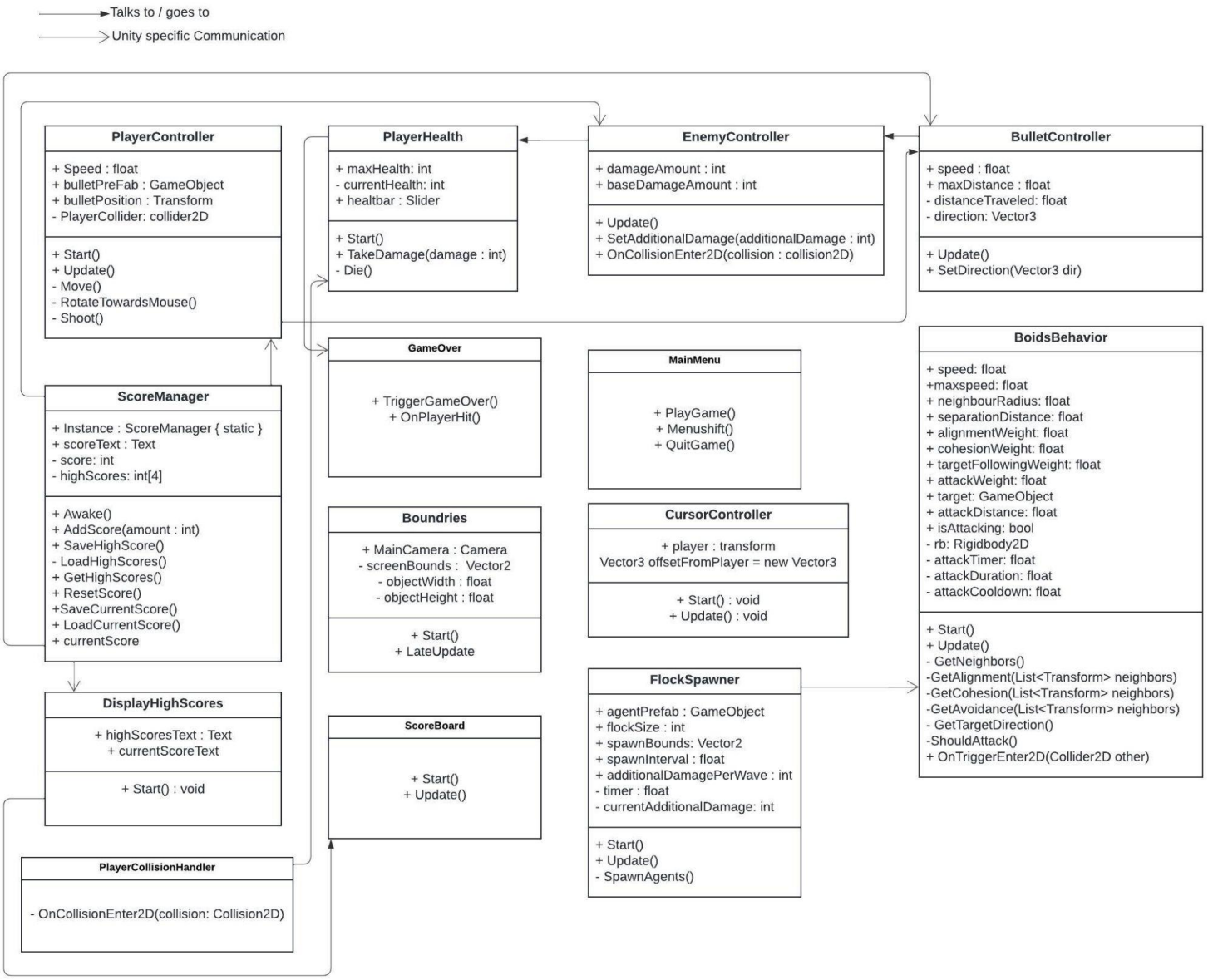


Figure 19: class / interaction / connection diagram

4.1 BoidsBehavior

This class simulates flock behaviour for AI-agents such as birds or fish. Here we will explain how each class contributes to the gaming experience, by creating realistic and dynamic herd group movements:

```
    public float speed = 2.0f;
    public float maxSpeed = 5.0f;
    public float neighborRadius = 2.0f;
```

```
    public float separationDistance = 1.0f;
    public float alignmentWeight = 1.0f;
    public float cohesionWeight = 1.0f;
    public float avoidanceWeight = 1.0f;
    public float targetFollowingWeight = 2.0f;
    public float attackWeight = 3.0f;
    public GameObject target;
    public float attackDistance = 5.0f;
    public bool isAttacking = false;
    private Rigidbody2D rb;
    private float attackTimer = 0f;
    private float attackDuration = 5f;
    private float attackCooldown = 10f;
```

These variables and attributes contributes in defining the behaviour of each boid-agent: '**Speed'** and **'maxSpeed'** sets the start and maximum velocity for the agents. They secure, that the agent's movements are dynamic and that it can vary inside these limitations. **'neighbourRadius'** and **'separationDistance'** decide how close the agents can be among each other, which is essential to avoid collisions and to create realistic herd behaviour. **'alignmentWeight'**,**'cohesionWeight'**,**'avoidanceWeight'** these behaviour components decide how much each behaviour component can interfere with the agent's movement. Here we can balance the simulation of the herd. **'targetFollowingWeight'** and **'attackWeight'** control how aggressively an agent follows a target or attacks, which is important for the interaction with the player. '**target**' and '**attackDistance**' Defines the target (the player) and the distance to initiate an attack, which gives a tactical parameter for the agent's behaviour. **'isAttacking','attackTimer','attackDuration','attackCooldown'** controls the agents 'attack behaviour' whether they are in 'attack mode' or not and how long time between the attacks.

```
void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        rb.velocity = Random.insideUnitCircle.normalized * speed;
        GameObject player = GameObject.FindGameObjectWithTag("Player");
        if (player != null)
        {
            target = player;
        }
        else
        {
            Debug.LogError("Player not found!");
        }
```

```
    }
```

‘GetComponent<Rigidbody2D>()’ gets and gives the **‘Rigidbody2D’** component, which is essential for handling movement related calculations for the agent. **‘Random.insideUnitCircle.normalized * speed’** sets the agents start speed for a random direction with the defined speed. This contributes to diversity in the agents' movement pattern. **‘GameObject.FindGameObjectWithTag(“Player”)’** finds the player object in the scene and sets it as the agent's target. This is important for the agent to be able to interact or react to the player's movement. Error handling with ‘**debug.LogError**’ ensures that there will be reported a mistake if the player object is not found, which is important under the development.

```
void Update()
    {
        List<Transform> neighbors = GetNeighbors();
        Vector2 alignment = GetAlignment(neighbors) * alignmentWeight;
        Vector2 cohesion = GetCohesion(neighbors) * cohesionWeight;
        Vector2 avoidance = GetAvoidance(neighbors) * avoidanceWeight;
        Vector2 targetDirection = GetTargetDirection() * targetFollowingWeight;
        Vector2 moveDirection;
        if (isAttacking)
        {
            moveDirection = targetDirection * attackWeight;
            attackTimer += Time.deltaTime;
            if (attackTimer > attackDuration)
            {
                isAttacking = false;
                attackTimer = 0;
            }
        }
        else
        {
            moveDirection = alignment + cohesion + avoidance + targetDirection;
            moveDirection = Vector2.ClampMagnitude(moveDirection, maxSpeed);
        }
        rb.velocity = moveDirection;
        if (!isAttacking && ShouldAttack())
        {
            isAttacking = true;
            attackTimer = 0;
        }
    }
```

The **‘update()’** method is where the agent's movement logic updates. **‘GetNeighbours()’** collects a list of nearby agents, which are necessary to decide the agent's movement in

relation to other agents. Calculations of movement directions. **‘alignment’:** Ensures that the agent guides itself in the same direction as the neighbours. **‘cohesion’:** Keeps the agent together with the herd. **‘Avoidance’:** Avoids collisions with the neighbours. **‘targetDirection’** Follows a specific target (The player) **‘moveDirection’** decides the actual direction of movement based on the factors above and the agent's present attack condition. **‘Attack mode’** updates and controls, if the agent should attack based on certain conditions. This method is about the agent's capability to react in real time in the game, and create a realistic and dynamic herd behaviour.

```
List<Transform> GetNeighbors()
    {
        List<Transform> neighbors = new List<Transform>();
        Collider2D[] colliders = Physics2D.OverlapCircleAll(transform.position, neighborRadius);
        foreach (Collider2D collider in colliders)
        {
            if (collider.transform != transform)
            {
                neighbors.Add(collider.transform);
            }
        }
        return neighbors;
    }
```

We use the **‘GetNeighbours()’** method to identify nearby agents.

We use **‘Physics2D.OverlapCircleAll’** to find all colliders in a given radius (‘neighbourRadius’) around the agent. Then we run through all the colliders and add their ‘transform’ to a list, if they aren't the agent themself. We use this to differ the agent from the rest of the objects nearby. This method is crucial for deciding the agent's interactions with its neighbours. And another step in calculating the herd behaviour.

```
Vector2 GetAlignment(List<Transform> neighbors)
    {
        if (neighbors.Count == 0) return transform.up;

        Vector2 avgDirection = Vector2.zero;
        foreach (Transform neighbor in neighbors)
        {
            avgDirection += (Vector2)neighbor.transform.up;
        }
        return avgDirection.normalized;
    }
```

The **‘GetAlignment()’** method calculates an average direction based on the agent's neighbours to create alignment. First we check if there are any neighbours, if not, return that

agents present upward direction as standard. Thereafter we sum the directions (**'up'** vector) of all the neighbours to find the average direction. And then we normalise the overall direction to secure, that the result is a unit vector. Here we ensure that the agents guide themself together in the same general direction as their neighbours.

```
Vector2 GetCohesion(List<Transform> neighbors)
   {  if (neighbors.Count == 0) return Vector2.zero;
      Vector2 centerOfMass = Vector2.zero;
      foreach (Transform neighbor in neighbors)
      { centerOfMass += (Vector2)neighbor.transform.position;}
      centerOfMass /= neighbors.Count;
      return (centerOfMass - (Vector2)transform.position).normalized;}
```

The **'GetCohesion()'** method calculates the centre of mass for the agent's neighbours to create cohesion. If there are no neighbours, the method then returns a zero vector, which indicates that there is no direction to follow to achieve cohesion. it sums the positions for all the neighbours to find the average point, which represents the centre of the mass for the group. It returns a normalised direction in the direction of this centre of the mass from the agent's present position. This contributes to keeping the herd together.

```
Vector2 GetAvoidance(List<Transform> neighbors)
   {
      Vector2 avoidance = Vector2.zero;
      foreach (Transform neighbor in neighbors)
      {
         Vector2 toNeighbor = (Vector2)neighbor.transform.position - (Vector2)transform.position;
         if (toNeighbor.magnitude < separationDistance)
         {
            avoidance -= toNeighbor.normalized / toNeighbor.magnitude;
         }
      }
      return avoidance.normalized;
   }
```

The '**GetAvoidance()**' method calculates an evasion direction to avoid collisions with the neighbours. It loops thru all neighbours and calculates the vector from the agent to every neighbour. If the distance to the neighbour is less than '**seperationDistance**' then add a direction, that pushes the agent away from the neighbour. It returns an avoidance vector, that secures that the agent keeps a safe distance to its neighbours and avoids collision. This method is crucial for a realistik separation between the agents.

```
Vector2 GetTargetDirection()
   {
      if (target == null) return Vector2.zero;
```

```
        return (target.transform.position - transform.position).normalized;
    }
```

The **'GetTargetDirection()'** method calculates the direction towards the target.
If there is not any target('**target**' is '**null**') it returns a zero vector, because there is no direction to follow. If there is a target, calculate the vector from the agent's present position to the target's position. This method is important to control the agent's movement towards the player.

```
 bool ShouldAttack()
    {
        if (Vector2.Distance(transform.position, target.transform.position) < attackDistance)
            return true;
        attackTimer += Time.deltaTime;
        if (attackTimer >= attackCooldown)
        {
            attackTimer = 0;
            return true;
        }
        return false;
    }
```

The **'ShouldAttack()'** method decides if the agent should shift to attack mode. First it checks if the agent is close enough to the player to initiate an attack.

```
 void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Bullet"))
        {
            Destroy(gameObject);
            Destroy(other.gameObject);
        }
    }
}
```

The **'OnTriggerEnter2D(Collider2D other)'** method handles collisions. When the agent collides with an object, it checks if it was tagged **"Bullet"**, if that's the case, both the agent '**Gameobject**' and the bullet **'other.gameobject'**. This method is crucial for the game's interaction mechanics, which handles what happens when an agent gets hit by a bullet.

4.2 PlayerController

```
public class PlayerController : MonoBehaviour
{
```

```
public float Speed = 5.0f;
public GameObject bulletPrefab;
public Transform bulletPosition;
private Collider2D playerCollider;
```

The **'PlayerController'** class initialises the player's movement controller. The variables **'Speed'**,**'bulletPrefab'** and **'bulletPosition'** makes it possible for us to control the velocity for the player velocity and shooting functionality in Unity's inspector. **'Speed'** decides how fast the player can move. **'bulletPrefab'** and **'bulletPosition'** refers to the projectile that we use and its spawn place.

```
void Start()
  {
    playerCollider = GetComponent<Collider2D>();
    transform.rotation = Quaternion.Euler(0, 0, 180);
  }
```

The **'Start()'** method initialises the components for the player character. The **'playerCollider = GetComponent<Collider2D>();'** gets and stores a reference to the players **'Collider2D'** component, which is necessary for physical interaction in the game, like collisions. **'transform.rotation = Quaternion.Euler(0, 0, 180);'** sets the player's startrotation. This line rotates the player 180 degrees around the z-axis.

```
void Update()
  {
    RotateTowardsMouse();
    if (Input.GetMouseButtonDown(0))
    {
      Shoot();
    }
    float x = Input.GetAxisRaw("Horizontal");
    float y = Input.GetAxisRaw("Vertical");
    Vector2 direction = new Vector2(x, y).normalized;
    Move(direction);
  }
```

The '**rotateTowardsMouse()**' is called to rotate the player, so it always is oriented to the mouse position on the screen, by mouse clicking (left button) fires a projectile by calling the method **'shoot()'**. This class allows the user to shoot. It collects input horizontal (left/right) and vertical (up/down) movement. This input decides the direction of the player movement. We use **'Move(direction)'** to make sure that the player movement is smooth and responds to the user's input.

```
void Move(Vector2 direction)
```

```
    {
        Vector2 min = Camera.main.ViewportToWorldPoint(new Vector2(0, 0));
        Vector2 max = Camera.main.ViewportToWorldPoint(new Vector2(1, 1));
        Vector2 playerSize = playerCollider.bounds.size;
        Vector2 pos = transform.position;
        pos += direction * Speed * Time.deltaTime;
        pos.x = Mathf.Clamp(pos.x, min.x + playerSize.x / 2, max.x - playerSize.x / 2);
        pos.y = Mathf.Clamp(pos.y, min.y + playerSize.y / 2, max.y - playerSize.y / 2);
        transform.position = pos;
    }
```

The **'Move()'** method handles the player's movement. First, the game's movement limits are set based on the camera's view. This ensures that the player stays within the playing area. Then we calculate the player's size to adjust the movement limits correctly. Then we update the players position based on direction and the velocity. This secures a smooth movement. Then we secure that the player does not move out over the screen by clamping the position within the calculated limits.

```
    void RotateTowardsMouse()
    {
        Vector3 mousePosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        Vector3 direction = mousePosition - transform.position;
        float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
        angle += 90f;
        transform.rotation = Quaternion.Euler(new Vector3(0, 0, angle));
    }
```

'RotateTowardsMouse()' this method rotates the player, så it's always oriented towards the mouse position. First the mouse's screen position is converted to a position in the game environment. Thereafter we calculate the angle between the player and the mouse, where the player gets rotated to look towards this direction.

```
  void Shoot()
    {
        GameObject bullet = Instantiate(bulletPrefab, bulletPosition.position, Quaternion.identity);
        Vector3 mousePosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        Vector3 shootDirection = (mousePosition - transform.position).normalized;
bullet.GetComponent<BulletController>().SetDirection(shootDirection);
    }
}
```

This class instantiates a new projectile by 'bulletPosition' with standard rotation (**'Quaternion.identity'**). Then we find the mouse position in the game world, and calculate the shooting direction from the player's position to the mouse. The calculated direction gets

transferred to the projectile with help from the **'bulletController'** to ensure that the bullet is moving in the desired direction.

4.3 BulletController

```
public float speed = 8f;
public float maxDistance = 10f;
private float distanceTraveled = 0f;
private Vector3 direction;
```

The variables for this class are important variables for the projectile movement. First we have **'speed'**, that decides the bullet velocity. Then we have **'maxDistance'** defines the maximum distance the projectile can travel. **'distanceTraveled'** keeps order on the distance the projectile has travelled. **'direction'** decides the direction that the projectile moves in.

```
void Update()
    {
        Vector3 newPosition = transform.position + direction.normalized * speed * Time.deltaTime;
        transform.position = newPosition;
        distanceTraveled += speed * Time.deltaTime;
        if (distanceTraveled >= maxDistance)
        {
            Destroy(gameObject);
        }
    }
```

In the **'Update()'** the projectile's position updates by placing its velocity and direction to the present position, which moves the projectile forward. The total distance that the bullet travelled, updates. If this distance exceeds **'maxDistance'** the projectile gets destroyed. This class makes sure the projectile moves continually until it reaches its maximum distance, where after the get removed from the game.

```
public void SetDirection(Vector3 dir)
    {
        direction = dir;
    }
```

This method allows change of the projectile's movement direction. The method takes **'Vector3'** as a parameter that represents the desired distance. it updates the **'direction'** variabel to the new direction, which affects how the projectile moves towards in the **'update'** method.

```
void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Enemy"))
        {
            Destroy(collision.gameObject);
            ScoreManager.Instance.AddScore(1);
        }
        Destroy(gameObject);
    }
}
```

This method handles collisions. When the projectile hits an object marked as “enemy”, it destroys the enemy object, the score increases by destruction of an enemy. the projectile gets destroyed Independent of the collision types, which secures that the projectile gets removed from the game after it hits a target.

4.5 Boundries

```
public Camera MainCamera;
    private Vector2 screenBounds;
    private float objectWidth;
    private float objectHeight;
```

‘MainCamera’ is a public variable that refers to the head camera in the scene. This variable is assigned to the Unity Inspector. **‘screenBounds’** is a private variable that stores the screen's edges, these limits are used to limit the object's movement.**’objectWidth’** and **‘objectHeight’** are private varíables that keep half of the object's width and height. It's used to adjust the objects placement in relation to the screen edges to ensure it doesn't fall out of sight.

```
  void Start()
    {
dimensioner.
        screenBounds = MainCamera.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height,
MainCamera.transform.position.z));
SpriteRenderer-komponent.
        objectWidth = transform.GetComponent<SpriteRenderer>().bounds.extents.x;
objectHeight = transform.GetComponent<SpriteRenderer>().bounds.extents.y;    }
```

The start method calculates og sores the screens edges with help from **‘MainCamera’.** This is done by converting the screen's dimension to world coordinates. We also calculate half of the object's width and height. This anticipates that the object has a **‘SpriteRenderer’**

component, and is used to adjust the movements, so the object doesn't disappear from the screen at any point. This ensures the game doesn't fall out of the sight of view.

```
    void LateUpdate()
    {
            Vector3 viewPos = transform.position;
     betragtning.
       viewPos.x = Mathf.Clamp(viewPos.x, screenBounds.x + objectWidth, screenBounds.x * -1 -
objectWidth);
       viewPos.y = Mathf.Clamp(viewPos.y, screenBounds.y + objectHeight, screenBounds.y * -1 -
objectHeight);
       transform.position = viewPos;
    }
}
```

In the **'LateUpdate()'** the **'Boundries'** class the object's position gets adjusted to keep it inside the screen borders. It stores the object's present position in **'viewPos'** and it adjusts the **'x'** and **'y'** positions with use of **'Mathf.Clamp'** to secure, that the object stays inside the calculated screen limits taking its size into consideration. It also updates the object's position to 'viewPos' after adjustment. This method ensures that the object is always in sight for the player and doesn't move out of the screen.

4.6 flockSpawner

```
public class FlockSpawner : MonoBehaviour
{
   public GameObject agentPrefab;
   public int flockSize = 20;
   public Vector2 spawnBounds = new Vector2(10f, 10f);
   public float spawnInterval = 20.0f;
   private float timer;
   public int additionalDamagePerWave = 5;
   private int currentAdditionalDamage = 0;
```

```
  {
      SpawnAgents();
   }
   void Update()
   {
      timer += Time.deltaTime;
      if (timer >= spawnInterval)
      {
         SpawnAgents(); // Replace with your actual enemy spawning method
```

```
            currentAdditionalDamage += additionalDamagePerWave;
            timer = 0f;
        }
    }
    {
        for (int i = 0; i < flockSize; i++)
        {
            Vector2 spawnPosition = new Vector2(Random.Range(-spawnBounds.x, spawnBounds.x),
Random.Range(-spawnBounds.y, spawnBounds.y));
Instantiate(agentPrefab, spawnPosition, Quaternion.identity, transform);
        }
    }
}
```

‘Start()’ initialises the herd by calling **‘SpawnAgents()’**, which creates the given amount of agents.**(‘flockSize’)**. **‘Update()’**, handles the time based spawning of agents, when the **‘timer’** exceeds **‘spawnInterval’**, then a new herd gets made, and **‘currentAdditionalDamage’** increases, which increases the difficulty level in the game. **‘SpawnAgents()’** creates every agent by instantiating **‘agentPrefab’** on random positions within **‘spawnBounds’**.

4.7 EnemyController

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class EnemyController : MonoBehaviour
{
    public int damageAmount = 10;
    public int baseDamageAmount = 10;

    void Update()
    {
    }
    public void SetAdditionalDamage(int additionalDamage)
{
    damageAmount = baseDamageAmount + additionalDamage;
}
    void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            PlayerHealth playerHealth = collision.gameObject.GetComponent<PlayerHealth>();
            if (playerHealth != null)
            {
```

```
            playerHealth.TakeDamage(damageAmount);
        }
    }
  }
}
```

The **'EnemyController'** controls the enemies behaviour, **'damageAmount'** and **'baseDamageAmount'** are variables that control the damage an enemy can inflict on the player. The **'Update()'** method is empty, which indicates that the enemies behaviour doesnt update every frame, and further logic can be added later. **'SetAdditionalDamage(int additionalDamage)'** allows dynamic change of enemy damage based on game progression. **'OnCollisionEnter2D(Collision2D collision)'** handles collisions with the player. When the enemy hits the player, damage is inflicted via the **'PlayerHealth'** component.

4.8 PlayerHealth

```
public class PlayerHealth : MonoBehaviour
{
    public int maxHealth = 100;
    private int currentHealth;
    public Slider healthBar;
    void Start()
    {
        currentHealth = maxHealth;
        healthBar.maxValue = maxHealth;
        healthBar.value = currentHealth;
    }
    public void TakeDamage(int damage)
    {
        currentHealth -= damage;
        healthBar.value = currentHealth;
        if (currentHealth <= 0)
        {
            Die();
        }
    }
    void Die()
    {
        ScoreManager.Instance.SaveCurrentScore();
        ScoreManager.Instance.SaveHighScore();
        SceneManager.LoadScene(2);
    }
```

```
}
```

The **'PlayerHealth'** class handles the players health status. **'maxHealth'** and '**currentHealth'** control the players maximum and actual health. **'Healthbar'** is a UI component that shows the players present health status. **'Start()'** puts **'currentHealth'** to **'maxHealth'**, and then the health bar gets initialised. **'TakeDamage(int damage)'** reduces **'currentHealth'** and updates the health bar. If health reaches 0, **'Die()'** is called. '**Die()'** stores the present and highest score and reads the new scene (game over scene)**.**

4.9 ScoreManager

```
public static ScoreManager Instance { get; private set; }
  public Text scoreText;
  private int score = 0;
  private int[] highScores = new int[4];
```

'Instance' is a static variable that ensures that there only exists one instance of **'scoreManager'. 'scoreText'** is a UI component that shows the actual score. **'Score'** is a private variable that keeps track of the player's score. 'highScores' is an array that stores the highest score, with room for 4 values.

```
void Awake()
  {
    if (Instance == null)
    {
      Instance = this;
      DontDestroyOnLoad(gameObject);
      LoadHighScores();
    }
    else
    {
      Destroy(gameObject);
    }
  }
```

In the **'Awake()'** method, we first check if **'instance'** is **'null'.** If that's the case, then put Instance to this instance of **'scoreManager'**. We use, '**DontDestroyOnLoad(gameObject)**' so the object doesn't get destroyed, when the scene changes. Then the '**LoadHighScores()**' gets called, to load the stored highest scores, in the beginning of the game. If **'Instance'** is not **'null'**,we destroy the present GameObject, which prevents creation of more instances of **'scoreManager'.**

```
public void AddScore(int amount)
    {
        score += amount;
        scoreText.text = "Score: " + score;
    }
```

The **'AddScore(int amount)'** method handles the addition of points to the player score. When called, the '**score**' variable increases with the specified **'amount'**. It updates the **'scoreText'** text to reflect the new score. We use this class to track and show the players progress in the game.

```
public void SaveHighScore()
    {
        int[] allScores = highScores.Concat(new int[] { score }).OrderByDescending(n =>
n).ToArray();
        for (int i = 0; i < highScores.Length; i++)
        {
            highScores[i] = allScores[i];
            PlayerPrefs.SetInt("HighScore" + i, highScores[i]);
        }
        PlayerPrefs.Save();
    }
```

The **'SaveHighScore()'** method saves the highest scores. 'highScores' combines the present score to an array, and sorts them out. Then it updates the 'highScores' array with the highest score. Uses 'PlayerPrefs.SetInt' to save each highest score in Unity PlayerPrefs-system, so they can be retrieved and displayed in future game sessions. It calls **'PlayerPrefs.Save()'** to secure, that the changes that have been made inside playerPrefs, gets stored.

```
private void LoadHighScores()
    {
        for (int i = 0; i < highScores.Length; i++)
        {
            highScores[i] = PlayerPrefs.GetInt("HighScore" + i, 0);
        }
    }
```

The **'LoadHighScores()'** method loads the saved highscores. Where It loops through the 'highScores' array. For each position in the array, the stored score is retrieved from PlayerPrefs using the key "HighScore" plus the index (eg "HighScore0", "HighScore1", etc.). If there isn't a stored score, set the value to 0 as a standard.

```
public int[] GetHighScores()
    {
```

```
        return highScores;
    }
```

The **'GetHighScores()'** method returns the saved highscores. This method returns the highscores array. This makes it possible for other classes to get access to the stored highscores.

```
    public void ResetScore()
    {
        score = 0;
    }
    public void SaveCurrentScore()
    {
        PlayerPrefs.SetInt("CurrentScore", score);
        PlayerPrefs.Save();
    }
    public void LoadCurrentScore()
    {
        score = PlayerPrefs.GetInt("CurrentScore", 0);
        scoreText.text = "Score: " + score;
    }
        public int CurrentScore
    {
        get { return score; }
    }
}
```

'ResetScore()' resets the player score. **'SaveCurrentScore()'** Stores the current score in PlayerPrefs. This ensures that the score can be reloaded in future sessions. '**LoadCurrentScore()'** Loads the saved score from PlayerPrefs and updates the scoreText with this value. Used to display the saved score when the game starts.

4.10 DisplayHighScores

```
public class DisplayHighScores : MonoBehaviour
{
    public Text highScoresText;
    public Text currentScoreText;     void Start()
    {
            int[] highScores = ScoreManager.Instance.GetHighScores();
        highScoresText.text = "High Scores:\n";
        for (int i = 0; i < highScores.Length; i++)
        {
            highScoresText.text += (i + 1) + ". " + highScores[i] + "\n";
```

```
        }
        int currentScore = ScoreManager.Instance.CurrentScore;
        currentScoreText.text = "Your Score: " + currentScore;
    }
}
```

The **'DisplayHighScores'** class shows the highest score and the present score. **'highScoresText'** and **'currentScoreText'** are UI-components that show the scores. **'Start()**' retrieves and displays the highest scores from the **'ScoreManager'**. Each score is added to **'highScoresText'** with a number and a new line. The current score is also retrieved from the **'ScoreManager'** and displayed in **'currentScoreText'**.

4.11 MainMenu

```
  public void PlayGame(){
      ScoreManager.Instance.ResetScore();
      SceneManager.LoadScene(1);
   }
      public void Menushift(){
      SceneManager.LoadScene(0);
   }
   public void QuitGame(){
      Application Quit;
   }
}
```

The 'MainMenu' class controls the functionality of the main menu. 'PlayGame()' starts the game by loading a new stage (with index 1) and resets the score by calling **'ResetScore()'** from **'ScoreManager'**. **'Menushift()'** allows shift to another menu or scene (here with index 0). **'QuitGame()'** closes the application. **'Application.Quit()'** is used in the 'QuitGame()' to close the application correctly.

4.12 GameOver

```
public class GameOver : MonoBehaviour
{
   public void TriggerGameOver()
   {
      SceneManager.LoadScene(SceneManager.GetActiveScene().name);
   }
```

```
    public void OnPlayerHit()
    {
        TriggerGameOver();
    }
}
```

The **‘GameOver’** class handles the end of the game. **‘TriggerGameOver()’** loads the present scene again, which is used to show the **‘Game Over’** screenshot. The **‘OnPlayerHit()’** method designed to be called when the player gets hit (by an enemy), which trigger the games end thru **‘TriggerGameOver()’**

5. User Manual

This section serves the purpose to guide people through this game's user controls. Unlike game veterans who already are accustomed to the world of video games it comes rather naturally for them like breathing. User controls differ according to the genre of the game, such genres in modern games today could be FPS (First (Point of view) Person Shooter), MMORPGs (Massive Multiplayer Online Role Playing Game), Soulslike (High difficulty), Rougelike (Dungeon crawler), etc.

The controls of the player in StarSpeed (our game) is bound to typical WASD and arrow key setup. The WS keys and ↑↓ arrow keys move the player up and down in the map. While the AD keys and ←→ arrow keys move the player left and right in the map. The bullets are fired from the front and the ability to shoot bullets is bound to the left mouse button (LMB). While the way to rotate the ship’s direction is to move the mouse around.

6. Testing

While developing the game, we would at the same time do a debugging of the code to check whether there are any anomalies with the new implementations to the main source code or other unusual clash of the intended behaviour.

Unity has an inspector mode, which allows the developers (users) to change the public variables and then do a test run of the program.

6.1 Player Testing

We made an online survey, which we got family and friends to answer after trying out the game for us. We wanted to see how different people experienced the game, and therefore chose people from all different ages and experience with games. We also made sure to tell them the answers were completely anonymous, so we would get honest answers.
The survey is as follows (*figure 20*).

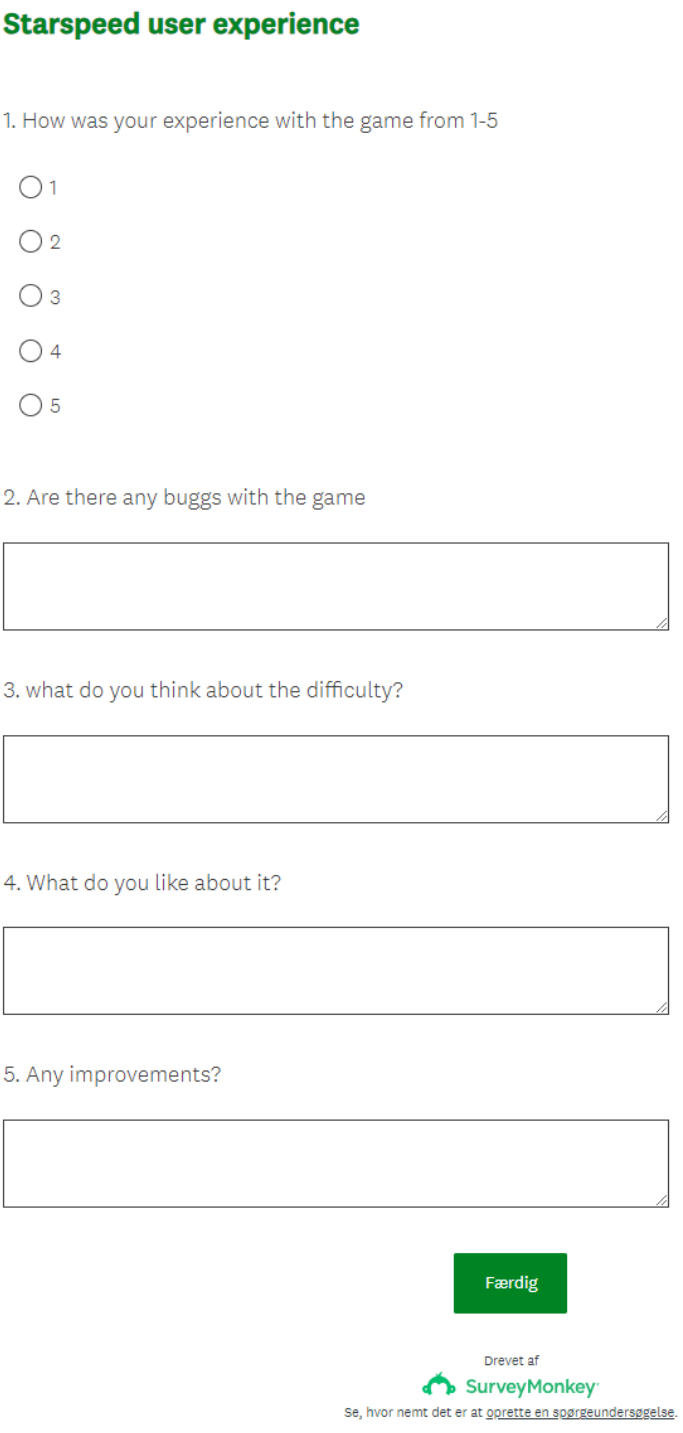


Figure 20: Survey

The feedback from the survey revealed encouraging results. We used 1-5 scale to gauge the games entertainment value, with 1 being lowest and 5 highest. The ratings were distributed as follows(*figure 21)*: 8.33% gave rating of 1, 0% gave a rating of 2, 50% gave the game a

rating of 3, then 33.33% gave the game a rating of 4 and lastly 8.33% gave a score of 0. This indicates that I found that the game was fun.
Regarding gameplay clarity, most comments suggested that the game was super easy to understand, and that they liked the UI aspect a lot.

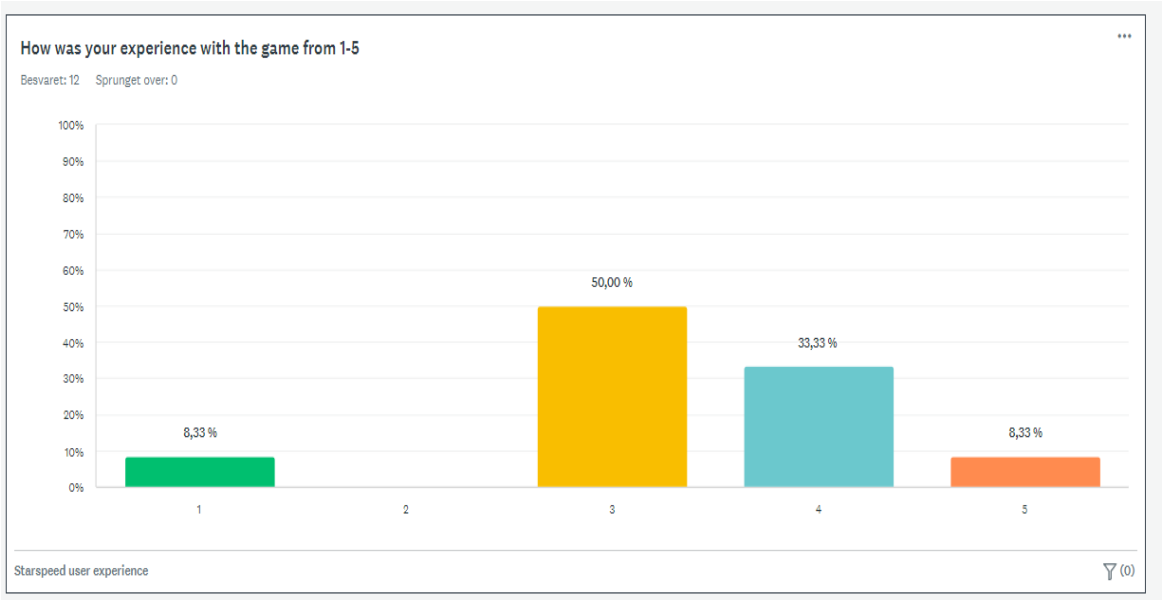


Figure 21: Survey ratings

Question number two was “Are there any bugs with the game” and some of the people who actually found some bugs as follows found that the shooting was off and sometimes the enemies move weirdly and get off the screen.
Question number 3 was “what do you think about the difficulty?” where we got about 70% answers saying the difficulty was very easy and then 30% of people saying it was too hard.
Question number 4 was “What do you like about it?” The answers here were that they mostly liked the UI.
Question number 5 was “Any improvements?” Here the answers were about a way to get health back when some is lost, what wave of enemies they were on, and some way to spice up the gameplay so it wouldn't be the same all the time.

Feedback analysis and interpretation
The survey provided invaluable insights into various aspect of the game which are discussed below:
Game Entertainment: The distribution of ratings indicates a pretty positive reception of the entertainment of the game. The majority of the players found the game to be pretty fun, this

suggests that while the game is quite engaging for most people there is still a lot of room for enhancing the experience.
Gameplay Clarity: The feedback was overwhelmingly positive in terms of gameplay clarity. Players praised the user interface (UI) for its ease of understanding, which is a critical aspect for the player engagement, especially for those that are a regular gamer. This aspect seems to be a strong point for the game.
Bugs and Issues: A notable concern was the detections of the bugs related to shooting mechanics and the movement of enemies. These bugs can impact the players experience and really need to be looked at. Fixing these will improve the gameplay experience.
Difficulty Level: The responses regarding the game's difficulty were mixed, with the majority finding it very easy and a smaller portion finding it too hard. This suggests a need for balancing the difficulty level to be better for a bigger range of players, implementing a difficulty setting could be a decent solution.
Player Preferences: Players expressed a strong liking for the game's UI, indicating that the visual and interactive elements of the game are well-received. This is an encouraging aspect that can be further developed to enhance player engagement.
Suggestions for improvement:Several constructive suggestions came from the survey. A clear indication of the current wave of enemies, and varied gameplay elements to keep the experience fresh and engaging. These suggestions are crucial for improving the game.

In conclusion, the survey has been a valuable tool in understanding the game from a players perspective and identifying areas for improvement. By addressing these aspects, the game could be alot better, for future revisions.

7. Discussion

Under the process of developing our StarSpeed game inspired from the game Space Invaders, our group had countless ideas we wanted to implement and different directions where we wanted to go. In this section we are gonna discuss the reasoning behind our choices, approach on the matters, and what we have accomplished with the finished developed product of our game.

7.1 Original plan vs final product

The original idea was to code a program, where we could make a game but with a new approach or concept, we had some ideas about which games we preferred, and which games that were possible to develop. We decided to focus on space invaders, and discuss how the enemies behaved. We wanted to change the behaviour of the enemies due to the reason that the aliens in Space Invaders have a simplistic movement and attack pattern.

Here we started discussing how we could implement an algorithm to control the enemies in a new and engaging way.

After brainstorming and discussing this idea with our supervisor, he suggested we take a look at Craig W. Reynold's boid model. The model introduces these behavioural characteristics of our enemy GO, by having them act accordingly with the 3 rules of separation, alignment and cohesion. In addition we wanted to have the enemies take advantage of the map, where the player is in the centre of the map and the enemy boids would have the player surrounded. While the boids would spawn infinite times until a threshold of points have been earned or to have a map clear condition of killing a defined number of boids.

From this point, we started designing the game, with the task of implementing the logic from the boid algorithm in our game design. One of the first ideas was to have several different types of enemies that had different settings on their parameters, so they all could respond with a different behaviour according to the players movements. Here we would have extra parameters for the enemy class such as how aggressive they should be, how fast they should react, how scared they are and when they should attack the player. So we had a lot of good ideas for the implementation, so we had to choose some of them off, but a lot of them are explained in the future work section.

The game we finally developed, is the player flying around in space, shooting the enemies while they following the player, responding to your interactions which is cool, because it gives a unique gaming experience each time you play the game, and you have to avoid the enemies as they come at you, and for each wave that spawns, you have to kill or avoid a whole group of enemies (20), and for every 20 seconds, the next group of enemies spawns, but this time they have more hitPoints, so colliding with the enemy will hurt you even more, than in the previous wave, the waves keeps coming and the enemies gets stronger, for each time they return, the waves are unlimited, so they will get stronger, for each wave.

In addition to the game, we also added a health bar so the player knows how much health they have left and we added a menu, with a start button and an exit button. Something that we have added, that really gave the whole game the good looking interface, and background pictures, and objects, was the pictures we created with mid journey, which was much easier than designing it yourself, we also found some sprites on the internet for the design of the player object and the projectiles the player shoots with. In this process Unity had some

benefits while designing the game, like the inspector window in Unity being able to adjust the public variables, so we don't have to access the scripts to make any changes in the parameters for the all the public variables in all the scripts, making it easier to change multiple variables, in the game.

7.2 Future work

At the start of the project, we had a myriad of ideas and direction we wanted to take but due to our own inexperience with game development and design, some of the features we wanted to implement did not make it into the current iteration of the game. In the case of us working on this project in the future, we will revisit the ideas of implementing the features, such as having different variants of maps, level increase, enemy types, (de)buffs, boss, and boss attack pattern.

7.2.1 Rules of Animals

We were thinking of implementing a feature that creates a passive/aggressive metre, which dictates whether boids would be either shying away from the player or taking an offensive attack stance. Another idea influenced by the animal kingdom was to set a predator status on the aggressive boids, which would attack the docile/shy and the player.

7.2.2 Map and enemies

An interesting feature that lies can passively affect the player and enemies would be the introducing of terrain types, which can be implemented in the maps. Not only can you add variation but also increase the difficulty by having different enemies with their own unique abilities, which forces the players to adjust to them. A couple of ideas could look like this:

A terrain where the visibility is limited and also known as fog or darkness. For the fog terrain the enemies would have increased attack range and damage just like a sniper unit. For the darkness terrain you can have enemies go invisible for a couple of seconds and if they reach the player they will explode and cause instant death.

Fiery terrain would be introducing DoT, where the player ship will take damage over time. The enemy bullets explode on contact dealing area damage (AoE), while it also adds a stack of burning state to the player and at the same time the enemies have burn resistance to counter the environment.

Icy terrain could introduce sturdier enemies that take longer to kill, while they shoot frost bullets that will add a freeze stack on the player. When the player gets x number of stacks of frost, the player will find themselves frozen and immobilised for a certain amount of time.

The increase in level after completing a map will result in spawning more enemies and increase their movement and attack speed. These numbers will need to be tweaked so it won't be too unbalanced.

7.2.3 Debuff, buffs, CC

Another approach to fine tuning the gameplay is the introduction of (de)buffs, which either affects the player positively or negative. In games they usually work as a status effect and could either increase or decrease the damage dealt, taken. Then we have crowd control (CC), which is a status abnormality rendering the target disabled or immobilising their movement. CCs are more well known in MMORPGs and DnD and can be categorised in 3 kinds of CCs, affecting movement, action/ability or forced action. CC's can determine the life of a target and therefore are important to avoid being affected by.

7.2.4 Shop and upgrades

A feature like a shop adds another layer of depth to the game. The items available in the shop can assist the player by upgrading the offensive and defensive traits of the player. The offensive upgrades could be increase in attack speed and damage, attack modifier like Area of Effect (AoE), bouncing effect in form of lightning conduction of nearby enemies, defence penetration as a counter for sturdier enemies, and tools that can assist the player in form of a self-shooting armament on the player or as a missile system the player can deploy it on the map. The defensive upgrades would work as a passive trait, and it will either increase the resistance to terrain elements of fire and ice or increase the visibility for fog and darkness. To have a shop we will have to introduce a form of currency for the game economy. That can either be resolved by having the enemies dropping game currency when killed that can be spent in the shop or having the player reaching a certain point threshold, which allows the player to choose an upgrade for free. Both methods require some knowledge of game economy design.

7.2.5 Boss

A classic feature like a boss is a way to implement a challenge to the player and usually a boss fight is seen as a climax, due to the indication that they always appear at the end of a game segment or as the very final of a game. A boss is the representation of a challenge due to being far stronger than the normal enemy encounters. The boss will usually have a theme and attack patterns corresponding to a specific theme. It could be an element like earth, wind, fire, ice, e.g., religion, legends, and myths. A boss fight has multiple phases which means that the first phase has 1 core game mechanic, and the boss will throw some simple attack patterns like swipes, hits, or throws at the player but starting after phase 1 the boss will suddenly add more game mechanics to the fight or completely change its behaviour and patterns. This is what makes a boss fight an exciting challenge for the player to overcome. An example of a boss fight could be that the boss spawns in the middle and the player has to damage the boss while avoiding the attacks.

The Boss attack patterns in phase 1 should be simple and introduce the main aspect like shooting destroyable orbs/bullets. Once in a the boss will shoot a ring wave as an attack, where there is a section of the ring that is broken. This serves as a safespace for the player and it has to adjust to the situation. In Phase 2 the boss gets stronger, faster and more aggressive. It will also shoot homing missiles/bullets at the player during the orb pattern phase, which have a mix of indestructible orbs in with the destroyable ones. The ring waves mechanic intensifies by shooting out in shorter intervals. At random times the boss will also start moving and try to aim and ram into the player to deal a large amount of damage. Lastly you can add a death phase where the boss drops all defences and go on the offence as a last resort or it can add oneshot mechanics to it.

8. Conclusion

Recollecting our experiences with the goal of learning about creating a game and implementing our ideas into the world of programming, we feel that the results of our work has produced a game that is playable on a prototyping level. We have drawn a lot of experience from using Unity’s game engine, and how to develop a game using their engine. The tests came back positive for the tests in focus, there were some minor errors, but nothing crucial in comparison with the feedback from the usertest. The game itself came out very

good, it was more or less the outcome we hoped for, where the enemy behaviours are simulating flock behaviour and are affected by the players interaction in the game. The game turned out with the twist we had hoped for in the beginning of the project by implementing the boid algorithm in the game.

References

. (2022, October 2). . - YouTube. Retrieved December 18, 2023, from https://assetstore.unity.com/packages/p/sci-fi-gui-skin-15606

About Git. (n.d.). GitHub Docs. Retrieved December 16, 2023, from https://docs.github.com/en/get-started/using-git/about-git

Adams, E., & Dormans, J. (2012). *Game Mechanics: Advanced Game Design*. New Riders.

Alaliyat, S., Yndestad, H., & Sanfilippo, F. (n.d.). *Optimisation of Boids Swarm Model Based on Genetic Algorithm and Particle Swarm Optimisation Algorithm (Comparative Study)*. The European Council for Modelling and Simulation. Retrieved December 6, 2023, from https://www.scs-europe.net/dlib/2014/ecms14papers/simo_ECMS2014_0062.pdf

Anderson, A., Gustavson, J., & Skarler, V. (2015, November). Space Invaders.

Boids-algorithm. (n.d.). Van Hunter Adams. Retrieved October 16, 2023, from https://vanhunteradams.com/Pico/Animal_Movement/Boids-algorithm.html

Fork a repository. (n.d.). GitHub Docs. Retrieved December 16, 2023, from https://docs.github.com/en/get-started/quickstart/fork-a-repo

Haas, J. K. (2014, Marts). *A History of the Unity Game Engine*.

Jarczyk, O., Gruszka, B., Jaroszewicz, S., Bukowski, L., & Wierzbicki, A. (2014, November). *GitHub Projects. Quality Analysis of Open-Source Software*.

LEE, T. Y. (n.d.). *Free Effect Bullet Impact Explosion 32x32 by BDragon1727*. BDragon1727. Retrieved December 18, 2023, from https://bdragon1727.itch.io/free-effect-bullet-impact-explosion-32x32

LEE, T. Y. (n.d.). *Free Effect Bullet Impact Explosion 32x32 by BDragon1727*. BDragon1727. Retrieved December 18, 2023, from https://bdragon1727.itch.io/free-effect-bullet-impact-explosion-32x32

Radhakrishnan, M. (2023, January). IS MIDJOURNEY-AI A NEW ANTI-HERO OF ARCHITECTURAL IMAGERY AND CREATIVITY?: AN ATYPICAL ERA OF AI-BASED REPRESENTATION & ITS EFFECT ON CREATIVITY IN THE ARCHITECTURAL DESIGN PROCESS. *Global Scientific Journals*, *11*(1), 94-104.

Repository limits. (n.d.). GitHub Docs. Retrieved December 16, 2023, from https://docs.github.com/en/repositories/creating-and-managing-repositories/repository-limits

Reynolds, C. W. (1987, July). Flocks, Herds, and Schools: A Distributed Behavioral Model. *ACM SIGGRAPH Computer Grapihcs*, *21*(4). https://doi.org/10.1145/37402.37406

Roeder, M., & Wolanski, J. (n.d.). *Space Invaders*. Wikipedia. Retrieved December 16, 2023, from https://en.wikipedia.org/wiki/Space_Invaders

Simple Spaceships | 2D Textures & Materials. (n.d.). Unity Asset Store. Retrieved December 18, 2023, from https://assetstore.unity.com/packages/2d/textures-materials/simple-spaceships-81051

Visual Studio Code. (2019). *Visual Studio Code*.

What Is a Kanban Board and How to Use It? Basics Explained. (n.d.). Kanbanize. Retrieved October 16, 2023, from https://businessmap.io/kanban-resources/getting-started/what-is-kanban-board