Purgatory
Project write-up and review

Sam Braham

Contents

Development

- **Overview**
- **Game Play**
    - **Player Controller**
    - **Walls**
    - **Enemies**
        - **Chaser and Hunter**
        - **Shooter**
    - **Spawner**
    - **Shooting and score**
        - **The Weapon**
    - **Display**
    - **Formal Review**
- **Menu systems**
    - **Main and Option**
    - **Influence of the Options**
    - **Black Box notes**
    - **Formal Review**
- **The Scoreboard**
    - **Development**
    - **Validation**
    - **Formal Review**

Post Development Testing and Balancing

- **Walkthrough of game**
    - **Comments of game**
    - **Solution to problems found**
- **Testing against Criteria**
    - **Solution to problems**
    - **Testing Video**

Evaluation

- **Evaluation against Aim**
- **Evaluation against Usability**
- **Further development**
    - **Potential Solution**
- **Limitations**
- **Conclusion**

Appendix

Bibliography

Introduction

This A-Level project addresses the problem of the need for computational entertainment in the form of gaming. The solution is a prototype of a top down shooter, where the player has a birds eye view of the arena and controls a character that fights against enemies that spawn regularly. Doing this increases their score which they can use to compare their skills to those of other players.

Analysis

Analysis - Problem Identification

Game development presents many possibilities for the programmer. From puzzles, challenging players cognitive ability, to reflexes, challenging player reaction times to stories, providing interesting and entertaining pieces of art, the possibility for games means there is much for a player to do. The intention is to design a top down shooter that will challenge the players' reflexes, strategy and awareness, as well as encouraging them to improve over time as the randomly generated enemies increase in difficulty. Their score will be recorded and saved so that they can try to beat their previous score, and other people's, to show their improvement in these skills. These problems and solutions are, therefore, suited to a computational solution owing to the modular nature of the game, through the use of abstraction and decomposition. This approach will allow for easy level creation and score keeping in addition to cleared and structured development by decomposing the problem down and abstracting away the areas that are unclear. An object oriented programming paradigm suits this style of programming as it makes it far more straightforward to create multiple enemies using classes and objects that interact with the player. In addition the very nature of the solution is dependent on a computer. The game will be shown to the player through a monitor and interacted with via a mouse and keyboard. As well as this, the complex decision trees that repeat many times a second, that use these inputs can be designed to function optimally in a computer system. Therefore the solution must be constructed through a computational methodology. To do this, the development will be carried out using the Unity IDE to help with Object Orientated Programing (OOP) and other features added by the use of the IDE.

Aim - To make a product that "challenges the players reflexes, strategy and awareness, as well as forcing them to improve over time"

The solution will be a Top Down Shooter with one room where enemies will spawn at a gradually increasing rate and level. There will be different types of enemies such as shooters, that use projectiles to attack, and chasers that will follow the player forcing them to move. The player will have a weapon to defend themselves. This weapon may vary, for example they could have a blade that is melee range but allows for faster movement or a rifle that has high damage and precision but slows movement. The player will get points for kills as well as a multiplier for their time alive that will go towards their score. This will be saved to their account name and where they can try and beat themselves and other players.

Depending on the responses from research of existing solutions and a public survey, features such as power ups and randomly generated obstacles could be added to make the solution more complex and interesting to the player. These features add more complexity, although are not essential for the final product and, therefore, may not ultimately be considered to be within the scope of the project.

Analysis - The Stakeholders

The solution is intended to be used by the gaming community of 12-18 year olds, mainly new and intermediate, due to the simple and repetitive nature of play. The solution will have an explanation as to how it is played, although the majority of the solution will be taught intuitively as the player moves through it. Each enemy type, for example, won't tell the player what it does, it will perform an action and the player must learn its patterns. Due to this gradual learning curve the solution would be best suited for gamers of the intermediate level as it may seem too sharp for new players, although this is less of a barrier to new players as it is a hurdle. Through development and testing, a good balance between different levels of teaching and intuition can be found to accommodate for the largest number of stakeholders.

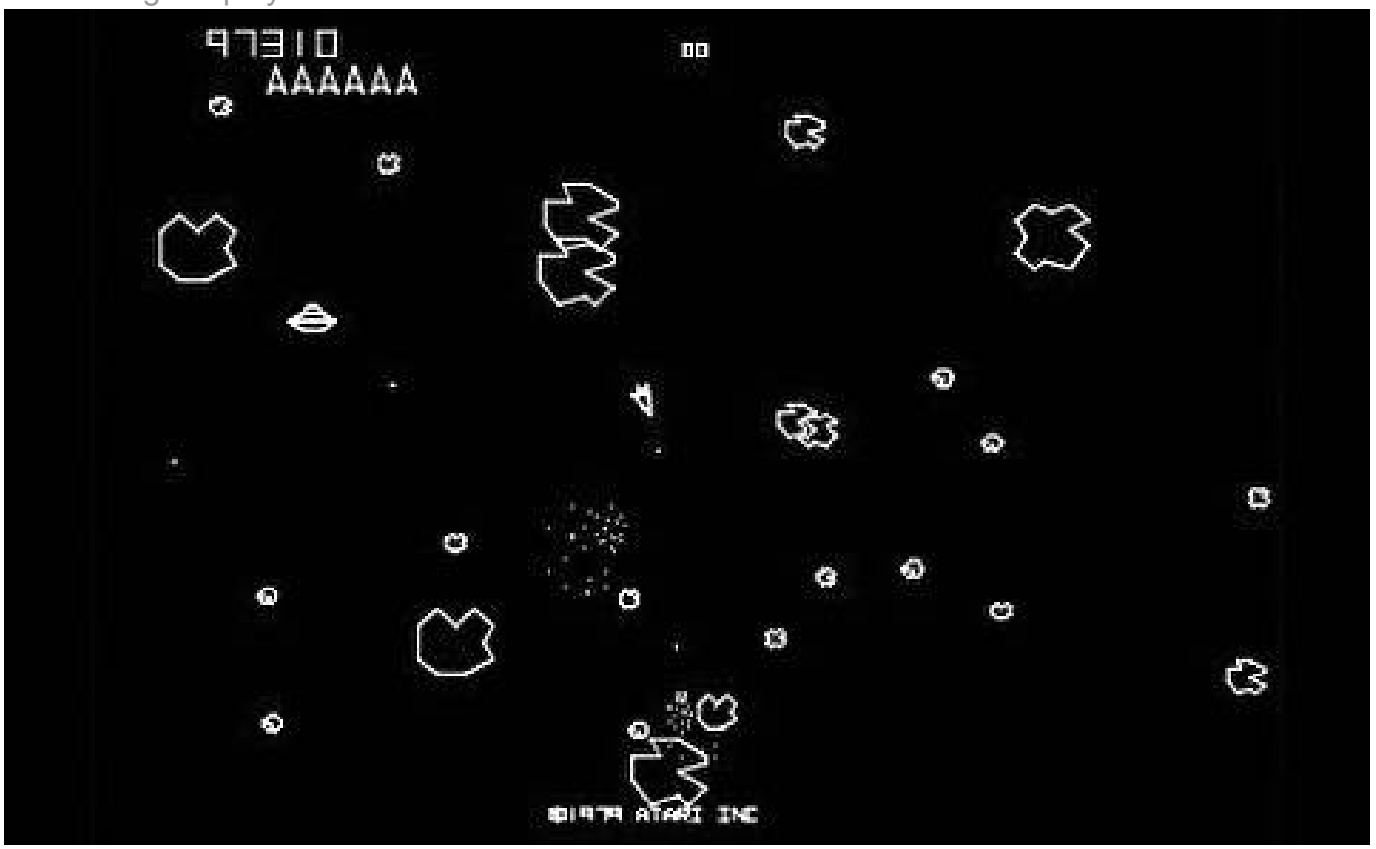
Analysis - The Research - Existing solutions

There are a wide range of Top-Down Shooters available (TDS) that vary in play and style. Many TDSs use a Rogue-like model in their game where players move through a procedurally generated world and try to get as far as possible before they die, at which point they are sent back to the start. Examples include Nuclear Throne and Enter The Gungeon. In both games the player moves through procedurally generated rooms fighting different and random enemies and unlocking new items before fighting bosses and moving on the new floor. These games show that variety in player actions and enemies are beneficial in creating an interesting game. The intention is to create a game with multiple types of enemies and, possibly, different weapons to create this interesting experience. The intention is not to create multiple floors/rooms in the game as this level of complexity would be out of the scope of the project and not necessary for the prototype.

Enter The Gungeon gameplay

Another style of TDS is the Arcade-like TDS, that presents the player with one room and constantly generates new targets. This is best shown by the game Asteroids. This endless TDS uses randomly generated targets to constantly provide threats for the player forcing them to be aware of their surroundings and be ready to adjust to the changing environment. These games show that one continuous level is engaging to a player as long as there is an appropriate and rewarding way of showing progress. Asteroids uses a showing system that gives points when the player destroys an asteroid allowing them to track their progress against others and themselves. My solution will be a continuous one level game and will use a scoring system to progress tracking, in the same way as in the Asteroids game.

Asteroids gameplay



Both types of game, Rogue-like and Arcade-like, use a health system that reduces every hit, with the player losing when their life total reaches zero. This is a simple way of determining how long a player will play for, punishing mistakes but allowing for them in, clearly showing how long they have left. Both games also use power ups in the game to add variety. Asteroids uses shields (blocking a hit for a short amount of time) and hyperspace (jumping the player to a random space on the map) and Enter the Gungeon uses items such as bombs and a box (player can hide in the box). All these add one time objects that the player can use when they think they need it, making them constantly aware that they have a one time answer to problems but there will always be a better time to use it. These types of features again are an addition to the game and (although highly changing the experience) would not be essential for the prototype.

Collision detection between parts of things are done differently between the two types of games. In Asteroids and older games collision detection was done when two colors touched, if a white space hit another white space that would cause the collision destroying the object. In newer games like Enter the Gungeon, object orientated programming allows for built in collision detection.

In conclusion, my research shows that creating a top down shooter with one room that spawns enemies would fit best, as it allows me to create an engaging and challenging experience without the time consuming repetitive workload needed for level creation. During the game, the player will be faced with different randomly spawning enemies and may have the choice of different weapons and/or items to use in the game to create a diverse and memorable experience. The solution will use a scoring system that rewards players based on time and kills to keep track of their progress in addition to a life system to determine how long the player plays for. This should result in an engaging and diverse experience without intense levels of complexity.

Analysis - The Research - Survey

A Survey was sent to around 80 individuals and received 53 responses. The intention of the survey was to clarify what exactly was desired by the stakeholders and gain a better understanding of the solution that was desired.

How much time do you play computer games (per day) *

- ○ Never
- ○ Up to half an hour
- ○ Up to an hour
- ○ Longer than an Hour

The first questions addressed the issue of people's knowledge in addition to determining how long people would play my solution for. If they answered never they would end the survey there. The other questions listed below address different areas of the solution and are explained in more detail along with the result data.

Have you ever played a Top Down Shooter *

- ○ Yes
- ○ No
- ○ Want to but never have

What keys would you prefer using *

- ☐ Arrow Keys
- ☐ WASD

Would you like background music / sound effects *

- [ ] Background music
- [ ] Sound Effects
- [ ] No

Would you want the difficulty to increase over time *

- ( ) Yes
- ( ) No

How many lives would you want before you loose the game *

- ( ) 1
- ( ) 2 - 4
- ( ) 5 - 10
- ( ) No life limit - just loose score

How do you want it to be scores *

- Just Kills (Score = Number of kills)
- Just Time (Score = time alive)
- Both (Score = Kills * Time alive)
- Other...

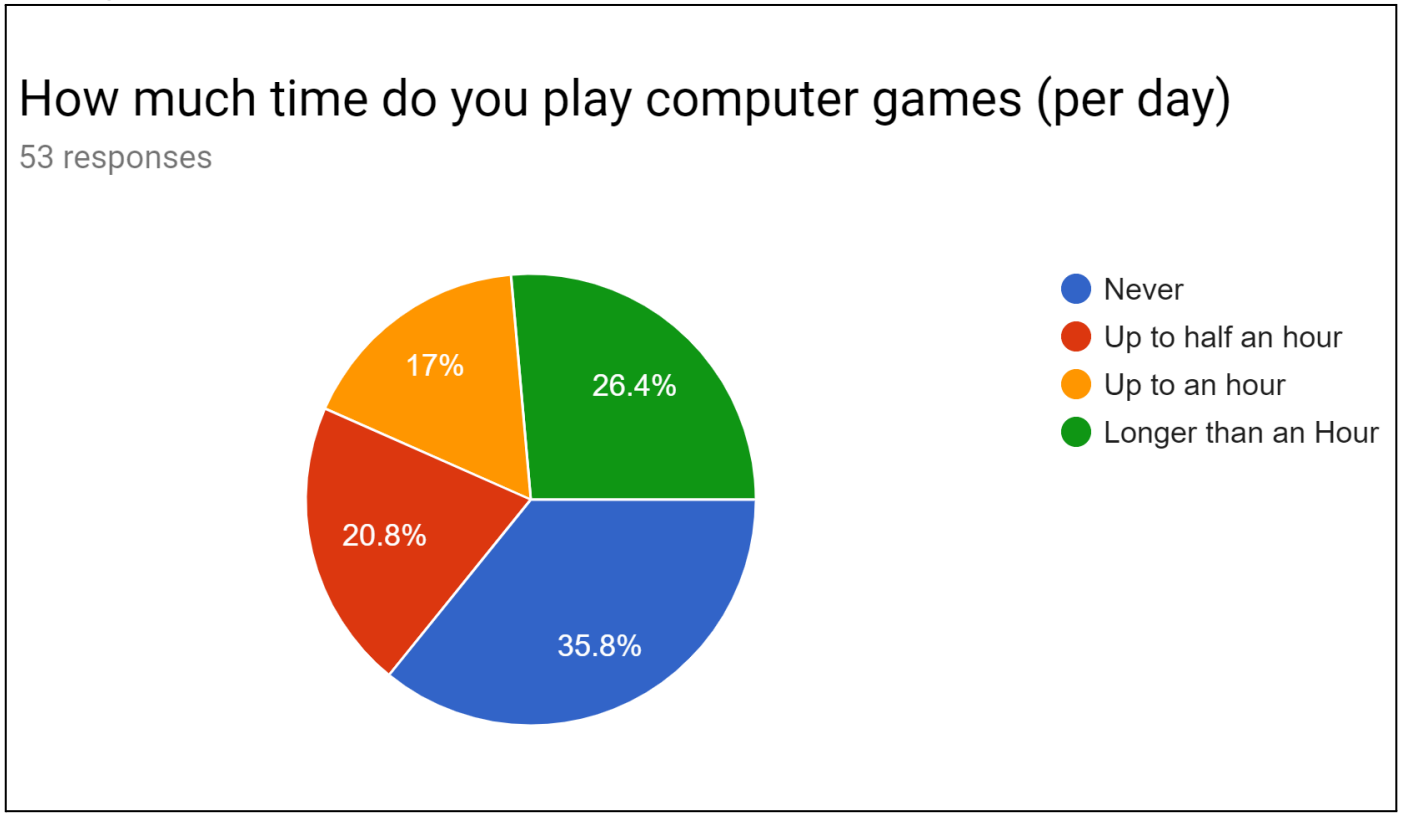
Would you want obstacles

- Yes
- Don't care

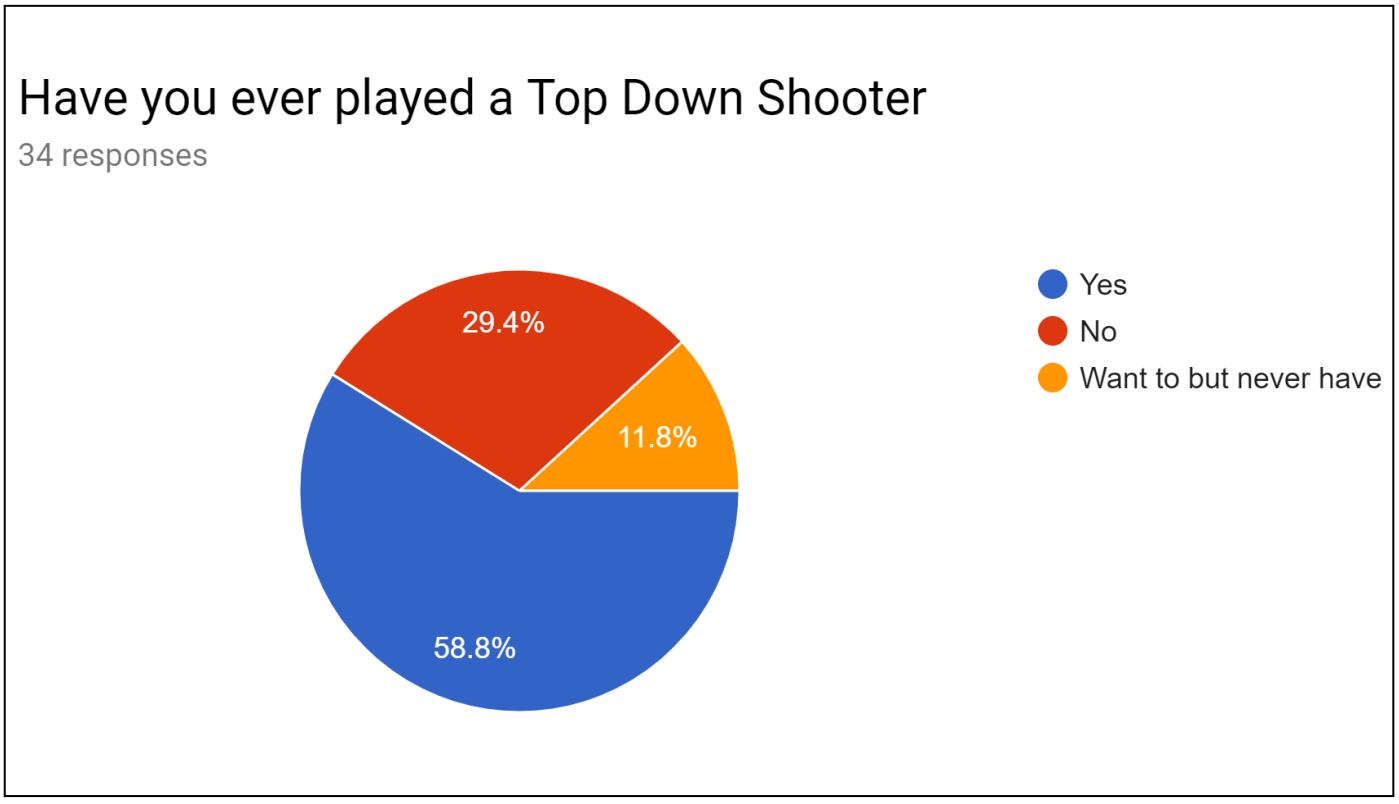
Would you want an 'End Game' *

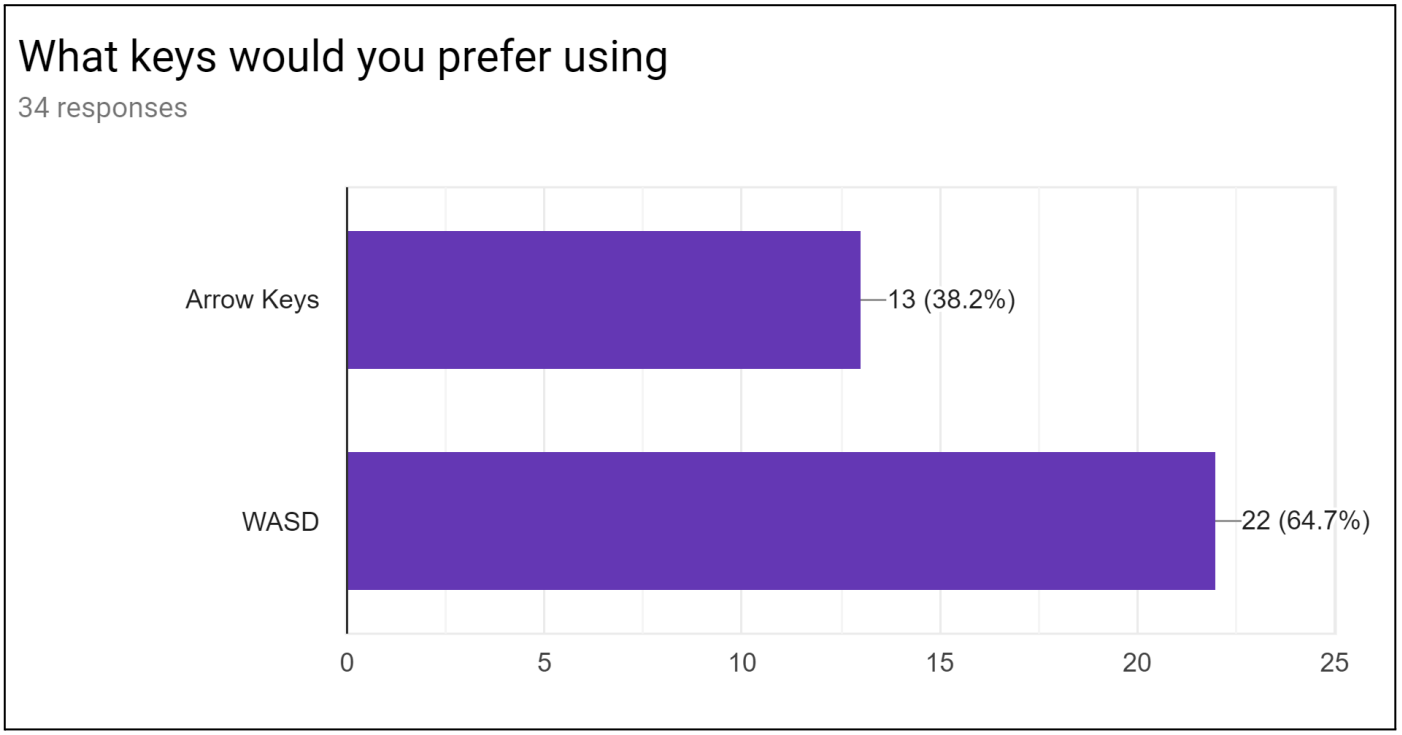
- No
- Yes - Ends after time / score
- Yes - Boss fight then ends
- Yes - Boss fight then continue
- Other...

Survey Results Data

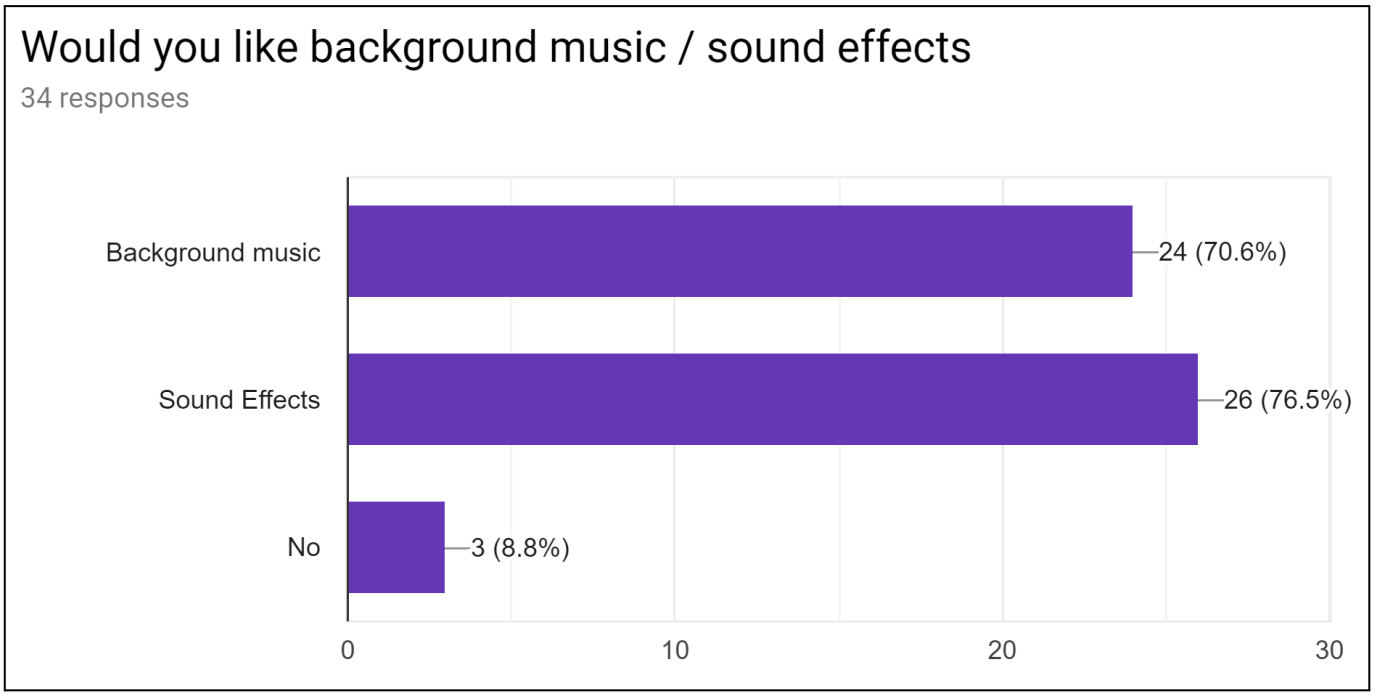


These two data points show that there is a large number of people wanting to play the game. Out of the original 53 names, 64.2% expressed an interest in games. Out of this group 70.6% expressed an interest in TDSs (45.3% of original 53). Only the people who showed an interest in games would continue the survey leading to the other questions having 34 responses

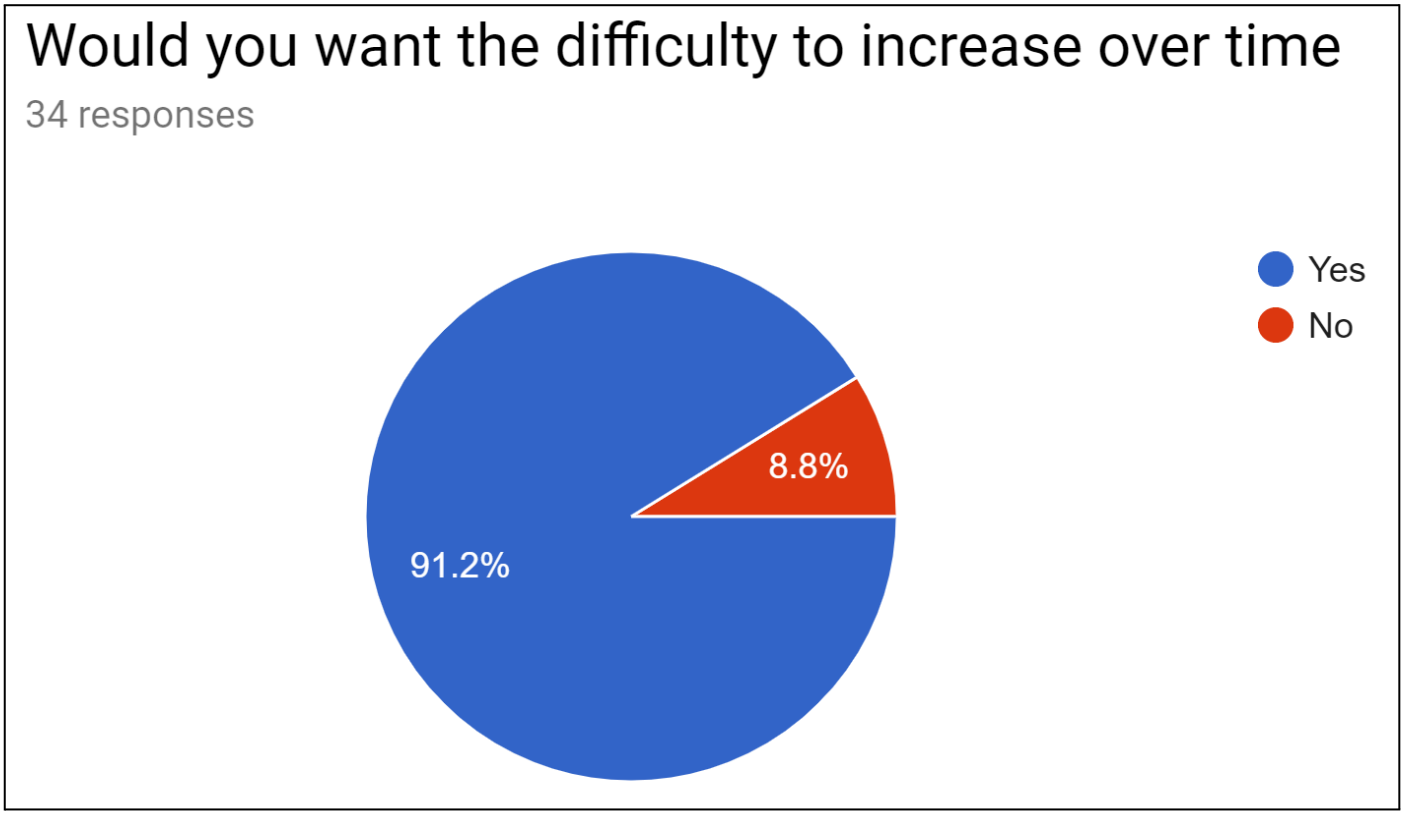




The next two questions allow the player to show their personal preference in games.
The questions on key bindings can change the feel of a game enough to make a difference, if you are not comfortable playing you won't like the game. Because of this, as well as the relative simplicity, the aim will be to add both options for the player.



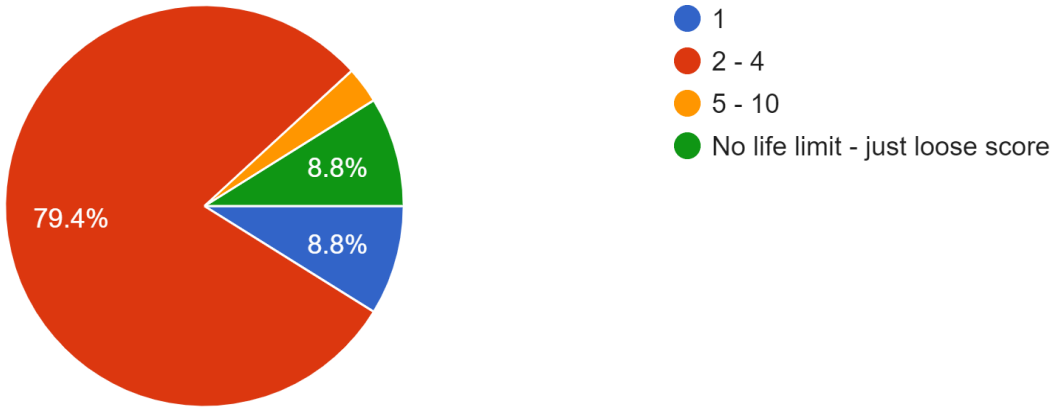
The questions on sound, although similar, present another issue of high complexity and difficulty to source. If there is time to commit to this in development, it could be added but due to the high complexity for little difference in game play, this means that this is not part of the scope of the program and will not be included in the prototype.



The question of difficulty changing had a clear response which is promising for the solution as the gradual change in difficulty is a crucial part of these kinds of games. The issue when it comes to it is getting the balance right between increasing fast enough to be challenging but not too fast to be overwhelming.

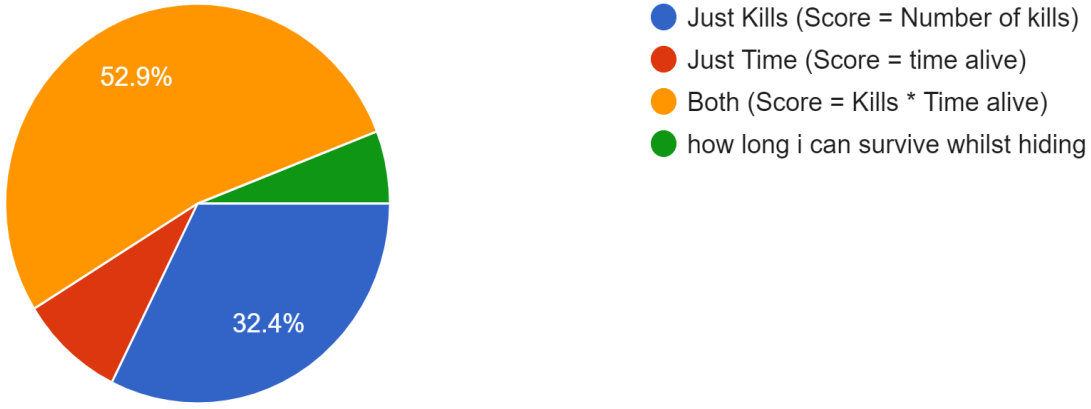
How many lives would you want before you loose the game

34 responses

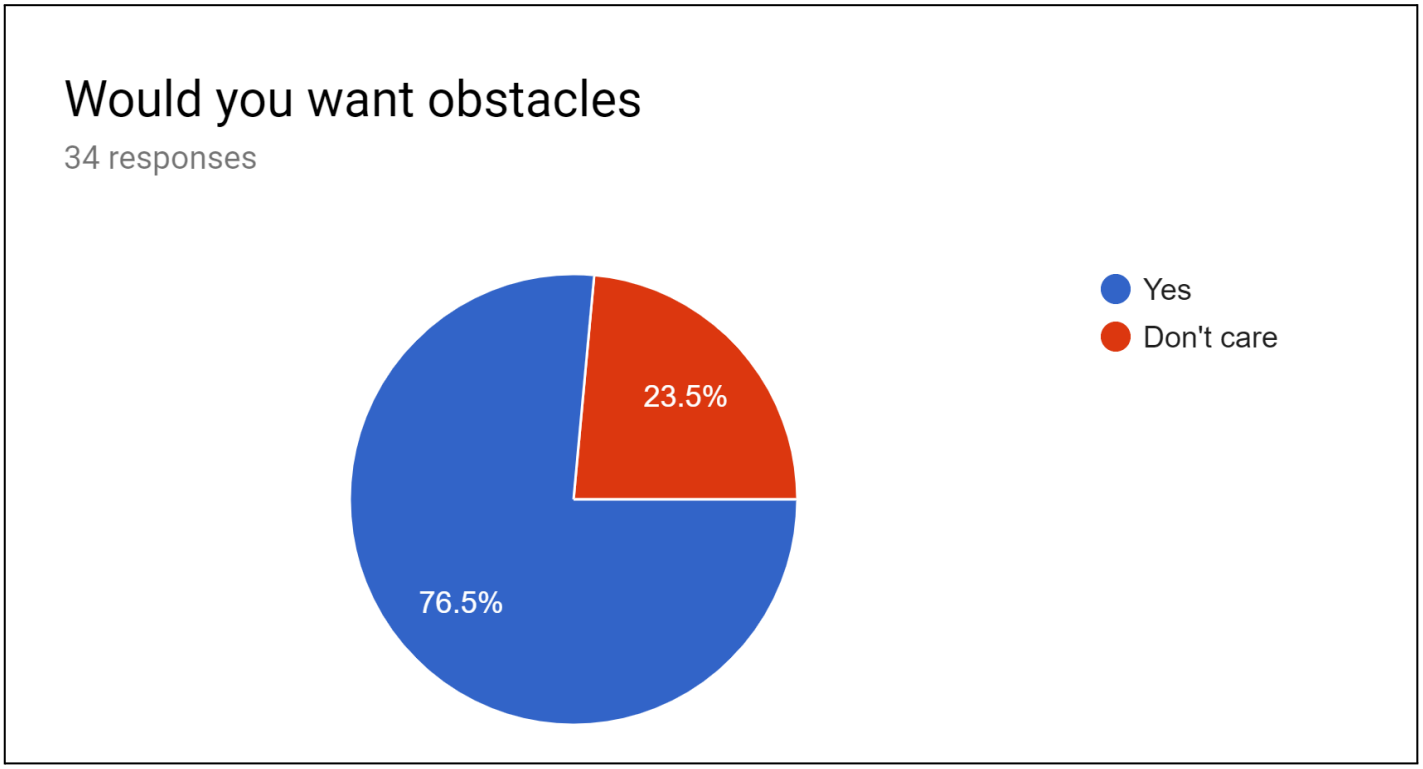


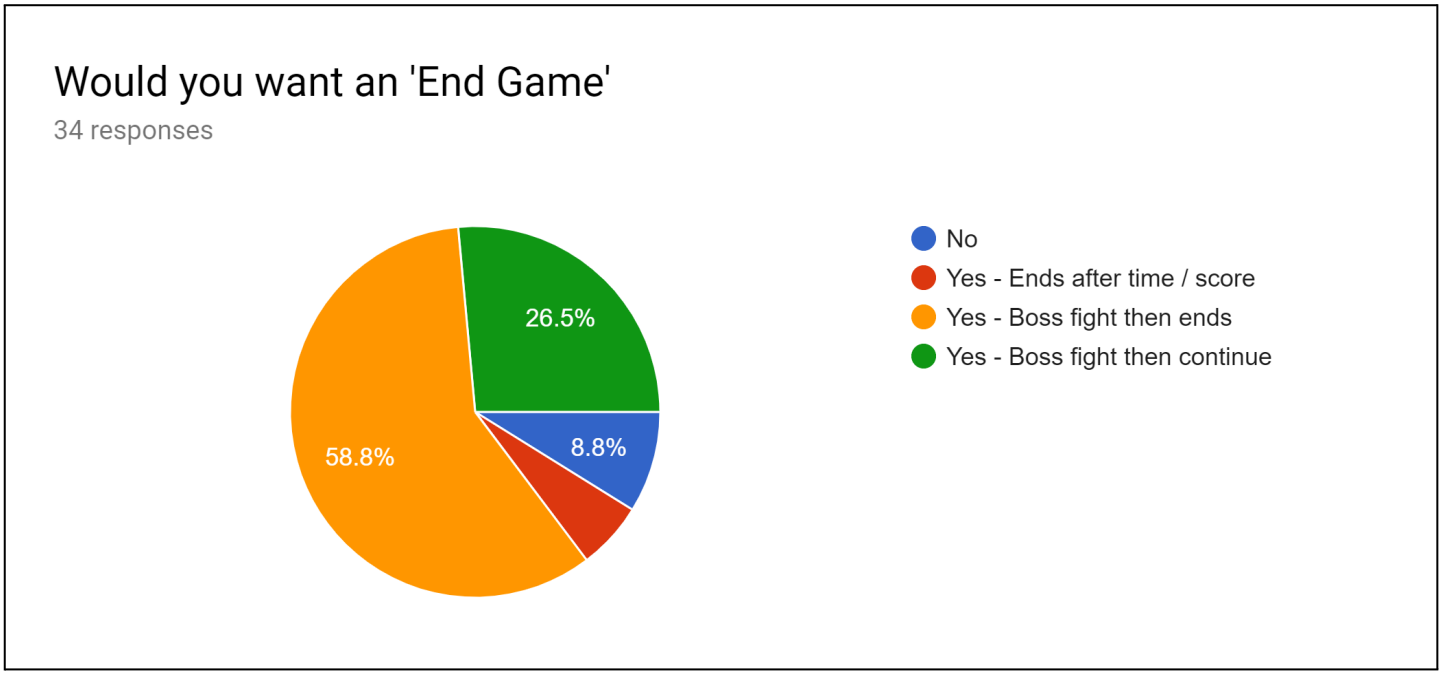
How do you want it to be scores

34 responses



This question along with the two above make very little difference to the final product. All show that the system that was planned in the initial research is supported by the survey (the game should contain obstacles etc.).
All these features will be added to the prototype.





The final question presents a developmental decision on the fundamentals of the solution. The majority opinion is for a final fight and then the game ending - noting that this is a small majority. This would be a significant and rewarding ending to the solution although it was decided that, due to the initial aim of the project, this was not the style of game that was intended for the final solution. A game with a final stage has less replayability than an endless shooter in addition to the fact, when a story isn't included, a final "Boss fight" can feel meaningless and empty as a final stage of the game. In conclusion, due to the initial aim to "challenge the players' reflexes, strategy and awareness, as well as forcing them to improve over time," the solution itself would be much better suited to an endless game that increases in difficulty. Going against market research but sticking closer to the initial aim of the solution.

Analysis - The Research - Conclusion

From the existing solutions:

- Continuous level with an increasing score - as shown by both solutions to create an increasingly difficult and challenging experience with low complexity
- Different enemies - as shown by the more roguelike solutions - as it allows for a more diverse experience, new enemies mean now challenges to learn and adapt to - a core part of the initial aim to "challenge the players reflexes, strategy and awareness, as well as forcing them to improve over time"
- Score board, rather than levels - as shown by the Arcade-like solutions. The score gives a point of reference to judge ability without having to build a multitude of different levels. In addition, creates a fairer experience for each player (levels present a decrease difficulty increase, score in on continuous a level presents a continuous difficulty increase. Less dramatic change in difficulty)
- When looking at the roguelike solutions, power-ups and different weapons add to the experience but, as shown by the simpler Arcade-like solutions, although are outside scope of the prototype.

From the survey:

- Obstacles would increase the experience - creating a more dynamic environment
- Multiple lives - giving player more chances when it comes to mistakes to ensure the aim of continuous learning is met, but not too many to still allow for a sufficiently challenging experience
- Dynamic score based on kills and time alive
- When it comes to the ending an endless game was decided upon, contrary to market research - The intention is to make a solution that tests the players ability when it comes to reflexes, strategy and awareness. Building a game with a boss fight that has an endpoint will go against this goal and so it will not have this.

Analysis - Features

From the the research conclusions the following design decisions have become clear

Essential:

- The player can move around the environment and interact with the enemies, mainly by using their weapons to eliminate them.
- The enemies will attack the player and damage their life total until they eliminate the player. The enemies must also spawn continuously for areas on the map at increasing rates.
- The player most earns a score from kills and time alive that is saved to a scoreboard.
- Score = Kills (a sum of the killed enemy scores) * Time was alive
- Different enemies add levels of difficulty as well as diversity in attacks to the player.

Potential:

- Different weapons such as melee and different project weapons - added decision making allows for different styles of play from fast and mobile to high damage and aggressive.
- Having multiple levels (obstacle layouts and spawning locations) so player have to adapt to new surroundings
- Items to add more score or power ups to increase complexity and reward styles of play.

Analysis - Limitations

Through the clarification of the solution, a number of limitations were found for both the player base and the solution itself:

- Due to lack of experience with the Unity IDE, there will need to be extra time given to understanding how the IDE works and how the tools can be used to solve the solution.
- The game is complex to play and requires a large number of rules that may or may not be learnt easily. In addition, the game assumes basic game intuition, some players will know how to play instantly others won't. It is unclear how much time if any should be spent on tutorial or teaching.
- The ability for the game to scale can be limited by the hardware of the user. As enemies spawn and the number of objects on the screen increase the processing needed of the game scales up such that late in the game it can be very demanding on RAM potentially limiting the player if there RAM is small and/or occupied with other processing.
- The game will only have one level/map due to developmental limitation of the prototype, this may be addressed in further development.

Analysis - Specifications

It is predicted that, due to the relative low complexity of the solution, it should not require much processing over Windows 10, a standard low spec OS. The minimum spec. is not high due to the simplicity of the program; presented below. If it is found to require more than this, then that will be appended in the evaluation of the program.

Processor:	Intel 8th generation processors (Intel i3/i5/i7/i9-7x), Core M3-7xxx , Xeon E3-xxxx, and Xeon E5-xxxx processors, AMD 8th generation processors (A Series Ax-9xxx, E-Series Ex-9xxx, FX-9xxx) or ARM64 processors (Snapdragon SDM850 or later)
RAM:	4 gigabyte (GB) for 32-bit or 16 GB for 64-bit
Hard drive space:	at least 128 GB for both 64-bit and 32-bit OS
Graphics card:	DirectX 9 or later
Display resolution:	800 x 600, a minimum diagonal display size for the primary display of 7-inches or larger.

Specs based on standard Windows 10 Recommended Specification

In addition to this, the development of the solution used the Unity development IDE. As the solution can be compiled this is not a requirement for playing the game.

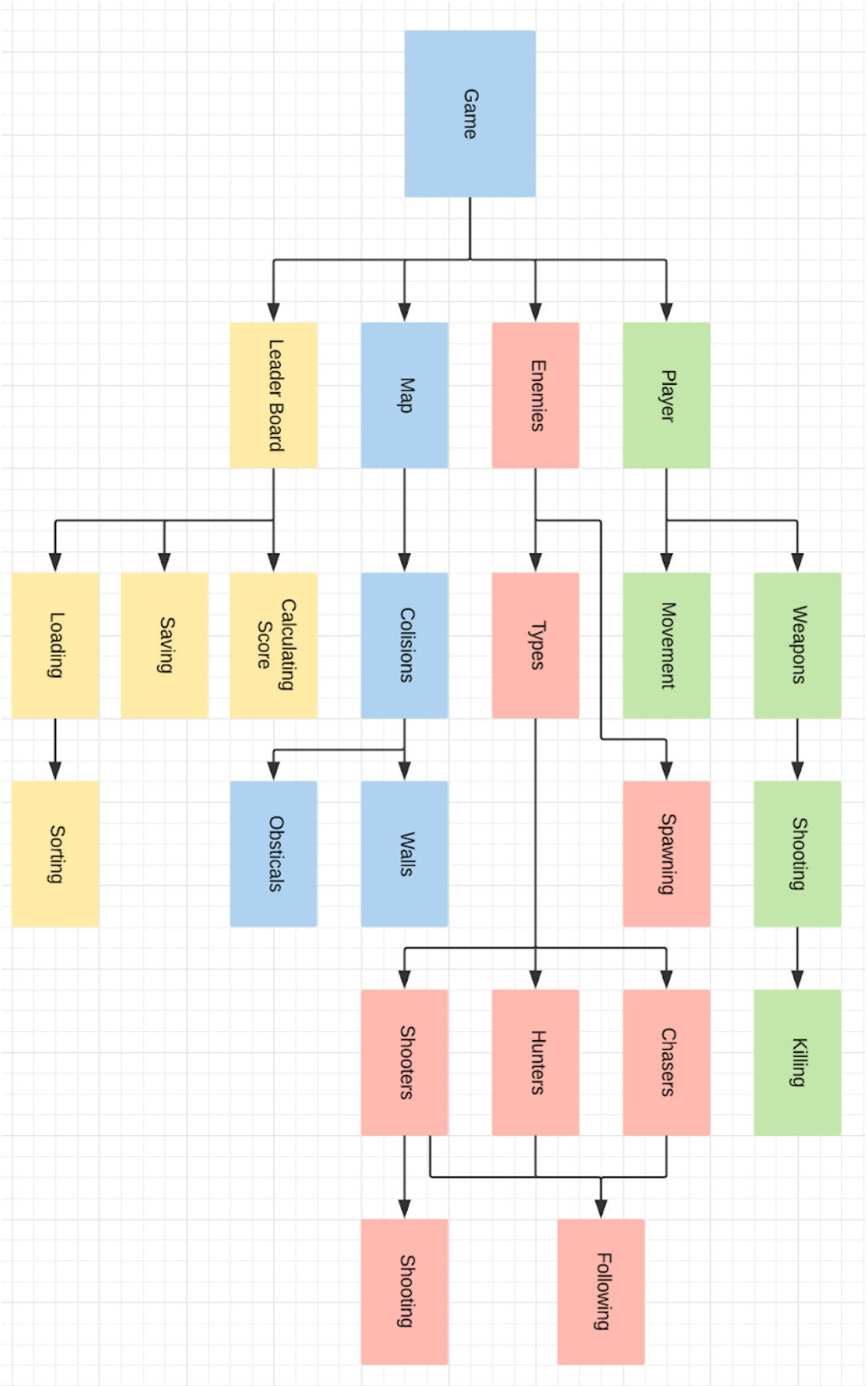
Analysis - Success Criteria

Game Play		
Player	The player must move in a two dimensional plane with equal movement in all directions	This is a standard feature of a top down shooter allowing the player to play the game
	The player must be able to attack with each weapon and damage the enemies	Allows the player to interact with the enemies and gain score
	The player must be killed (the game ends) when they run out of lives	Adds threat to the game to make challenge
Enemies	All enemies must be able to move towards the player in the intended manner	Adds threat to the game to make challenge
	All enemies must be able to attack the player in the intended manner to remove a life on a hit	Adds threat to the game to make challenge
	All enemies must be able to spawn at the intended time in the game (harder enemies are spawned later in the game)	Create constant threat and allow the game to go on continuously
	The rate of enemies increase as the game goes on (difficulty must increase)	Increase the challenge as the game goes on
Map	All moving entities should either be blocked or be destroyed when reaching the sides of the map. This is also true of any obstacles within the map	Create and interactive and varying play environment
Game Features		
Score	The Score must increase, additively, for every enemy killed (e.g. Score = Score + 1 when chaser killed)	Incentivises killing the enemies to teach the player not to let them build up
	The score must increase multiplicatively for time spent in the game alive, this should be done at the end of the game. (e.g. Score = Score * time when game ends)	Incentivises time in game to play longer
Saving	At end of game, must pick a username for their score to save	Allows the player to identify the score as theirs and find it later
	The users information along with their score must be saved permanently to the game.	Allows progress to be received the score later
Presenting	Any user should be able to see the top 20 or so scores and usernames, this would be sorted into highest score at the top of the list	Allows players to compare scores and try to beat other players or their own high score

Design

Design - Decomposition

This decomposition of the program shows the solution as a product of its parts, this will allow a structure to develop and through the development each section will be developed.



Design of Modules

In this section, each module will be designed individually. Its function in the program and then (if a specific solution is known) how it can be developed. If a specific solution is not known, the exact nature of it will be clarified through iterative development.
The development of the program will use the Unity IDE. This should function well with the object orientated programming paradigm that will be used. In addition, it can clarify and help solve some issues. All code in the project will be C#.

Design - The Player

The Player will be viewed as a small rectangle on the screen wielding a weapon. They will move around the screen using the arrow keys and aim using the mouse, that will be the extent of their actions. If they hit a wall or obstacle they will not be able to move past it and if they hit an enemy they will lose a life. Initially the player will have around 5 lives, testing is needed to determine exact value, and the player will die if their life total reaches 0 resulting in the game ending, their score being saved and them being offered the option to play again.

The player, although being the focus of the game, will have very little to do (in the initial iteration of the game) to ensure the game doesn't initially overwhelm the player.

Movement
The player will be considered as two separate fixed objects, the moving player character (PC) and the weapon that is always attached to the player although rotates to face the cursor. Due to this C# code can be used along with the built in Unity mechanics to move the player.

Using Unity's Update and Fixed Update function we can gather data from the player hitting the arrow keys and turn that into movement of the player character. This is done with Unity'sRigid Body component allowing it to move fluidly.

I can also normalize the inputs (only consider the direction, not the scale) to allow for equal movement in all directions, otherwise we would move twice as fast diagonally.

Weapon
The weapon is going to follow the mouse cursor around the screen and fire on a left mouse click. The weapon will be linked to the player character at all times. When the weapon fires, a projectile will be instantiated and move across the screen in a straight line in the same direction as the weapon. It will be deleted upon hitting an object, in turn deleting an enemy if that is what was hit. If it misses and hits all walls, the projectile is deleted and nothing else happens.

Fixing the weapon to the player is easily achieved with the linking feature in Unity. Making it follow the mouse cursor can be done by calculating the angle between the weapon and the cursor and then rotating it by that angle. Instantiating the projectile will be simply done by instantiating the prefabricated object at a point on the player with the same rotation as the player, then give it a velocity forward and use 2D collision components to detect if it hit anything.

During the research and analysis, the idea of different weapons was suggested. This will not be done as part of the prototype due to the fact that different weapons would not have significantly different components and features, mainly just different values in its variables and constants. Because of this, the idea will be moved into future development and not be done as part of this prototype.

Design - The Enemies

The enemies will be split into three different types of three scaling difficulties, this way in the later game the difficulty can increase rapidly without filling the screen with easy to fight enemies. All enemies will move towards the player, they will be stopped by obstacles and other enemies. The enemies will also die upon being hit with a projectile, noting that these projectiles may initially include those from the shooter class of enemies, although if this results in too much friendly fire then it will be removed.

All the enemies will be created at spawn points. There will be between 3 to 5 spawn points around the map where enemies will be created. They will be created procedurally at random, taking into account the score of the player and the enemy being spawned. Faster/more aggressive enemies will spawn later in the game. They will be instantiated every unit of time (e.g every second / every 5 seconds) from the start of the game. Late in the game multiple enemies will be spawned in different places.

Chasers - Easy
The chaser is the simplest enemy. It will move towards the player at a medium pace and attack them upon touch. After touching it will destroy itself

Shooters - Medium
The shooter enemy will shoot at the player with projectiles similar to the players. The enemy will move towards the player at a slow pace and stop before getting too close so as to not block other creatures and to be harder to hit. If the player is to move into the shooter it will also deal damage and destroy itself.

Hunters - Hard
The Hunter is the fastest and hardest enemy. It will move with a speed similar to the speed of the player. They will also die after hitting the player. They would also be harder to kill, taking two hits rather than one.

Design - The Map

The map will be relatively simple, just one map with boundaries and one or two simple obstacles. The obstacles will be impassable to all objects.

Collisions
The collision system will be done with Unity's two dimensional collider components and trigger. If between non-projectile objects, upon collision most things will stop until they move around the obstacle.

Design - The Leader Board / Score System

Every kill will be a score based on the enemies difficulty, and then at the end of the game the time alive will be multiplied to the score to give the players final score. The player then will have the option to save their score. They input a username, to tag their information, and a password, for security. Their score will then be saved in a file of all usernames, passwords and scores as well as the number of scores under the same username.

The Board

The player will have the option to look at past scores of other players and themselves. They can go to the leaderboard were their scores will be sorted into a list from high score to low score, along with the number that game was of that player, e.g.

SBraham (the username) - 124500 (the score) - 4 (my 4th game)

NB - Revision - Later on it was decided that the number of times a player has played the game would be too difficult to keep track of for multiple players with the style of game that was being developed, in addition the leader board will act much more like the old arcade games. At the end of the game, the player will enter a username (3-5 characters) and then that will be saved alongside your score and your ranking.

4th (4th place) - 124500 (the score) - BRA (the username)

The Calculation

Calculating the score will be a step done at the end of a game before saving it.

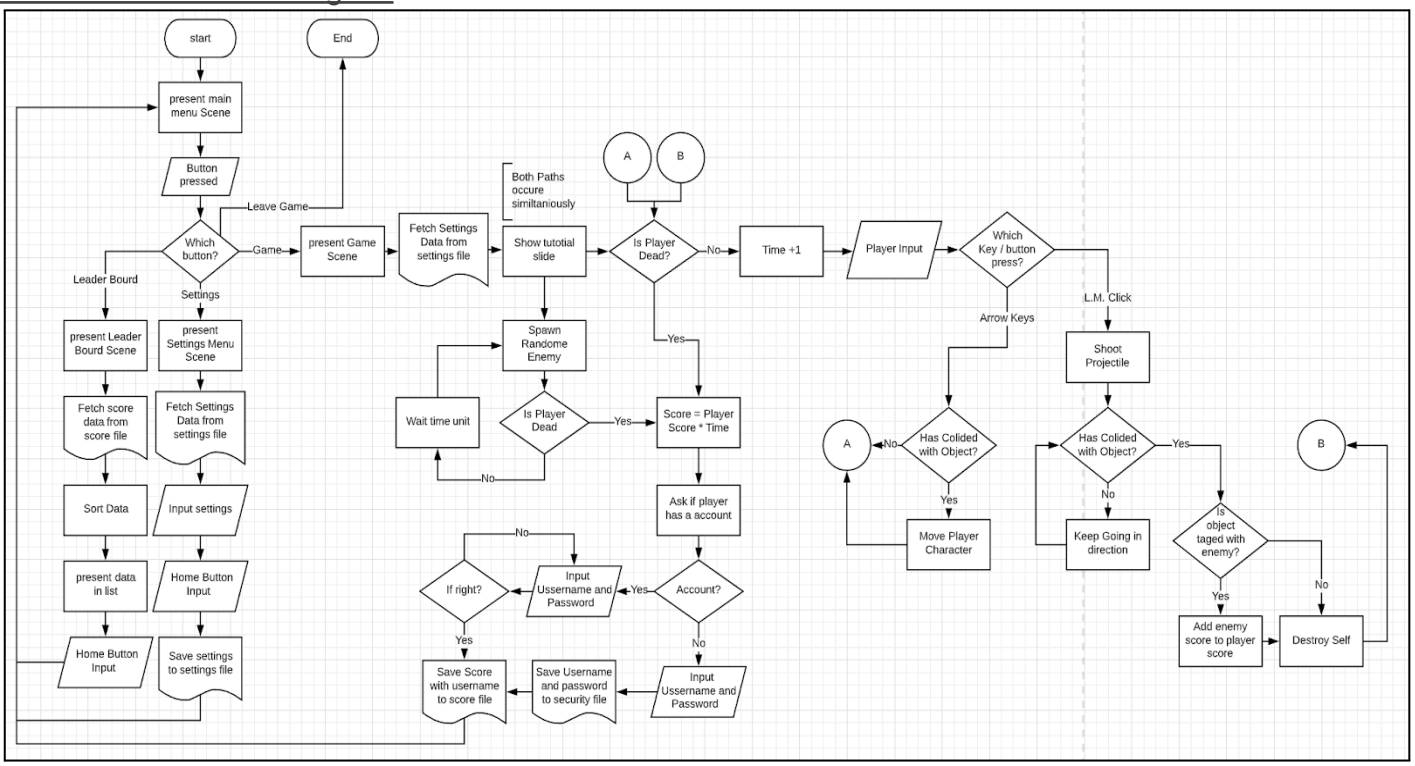
Score = Sum of the respective killed enemy scores (ChaserKills * 1 + ShooterKills * 3 + HunterKills * 5)
Multiplied by the time spent in game and alive

The Saving and Loading

To do this part of the project, the “StreamWriter” function form C# will be used that allows for the saving of data into a file. This will be done sy saving text into a Json file and as no personal information is being saved the file is not encrypted. This is another reason why the use of passwords was not used.

Design - Flow Charts

The main Flow of the Program

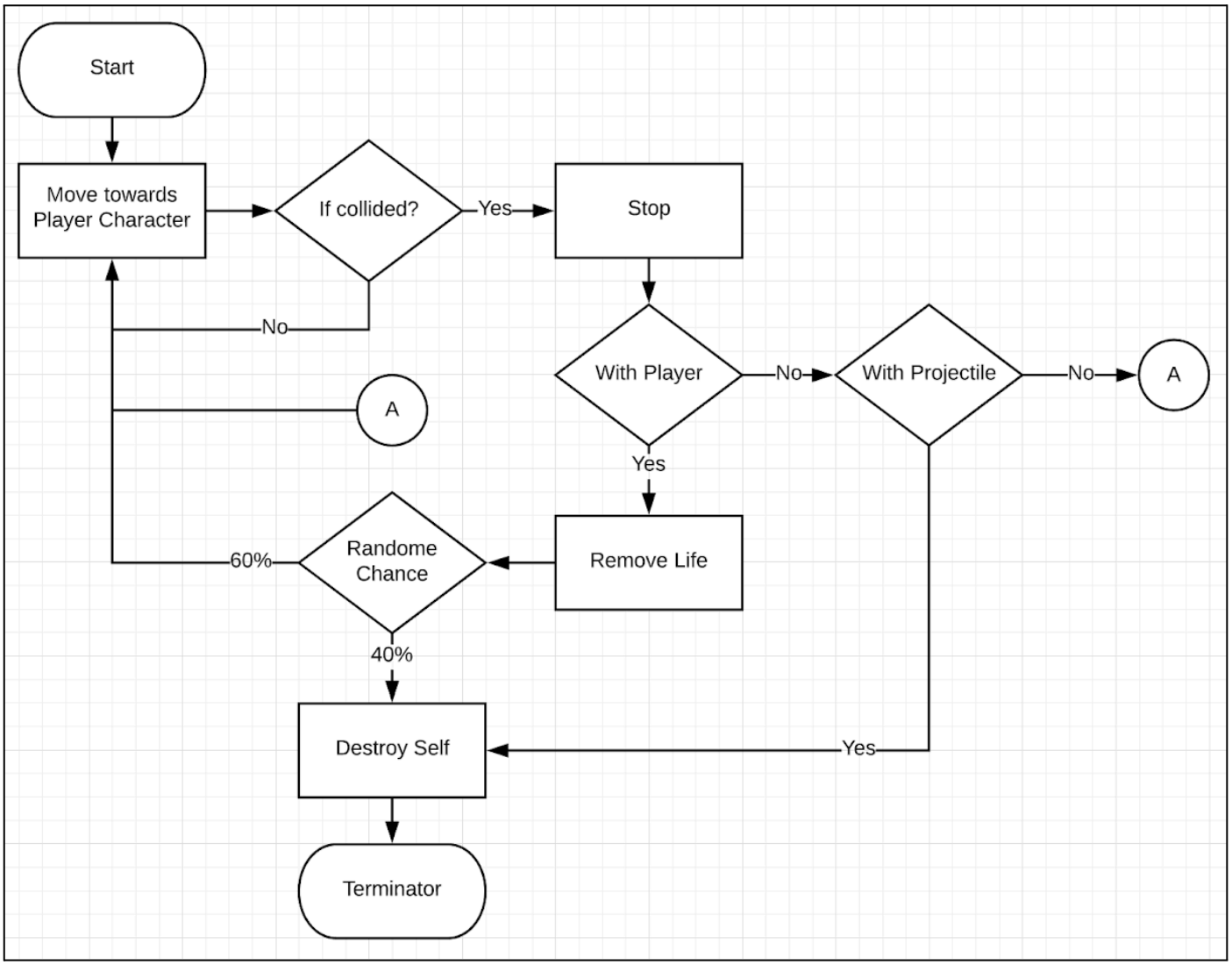


This is a massive flowchart and because of this a blown up copy of this is included on the following two pages.

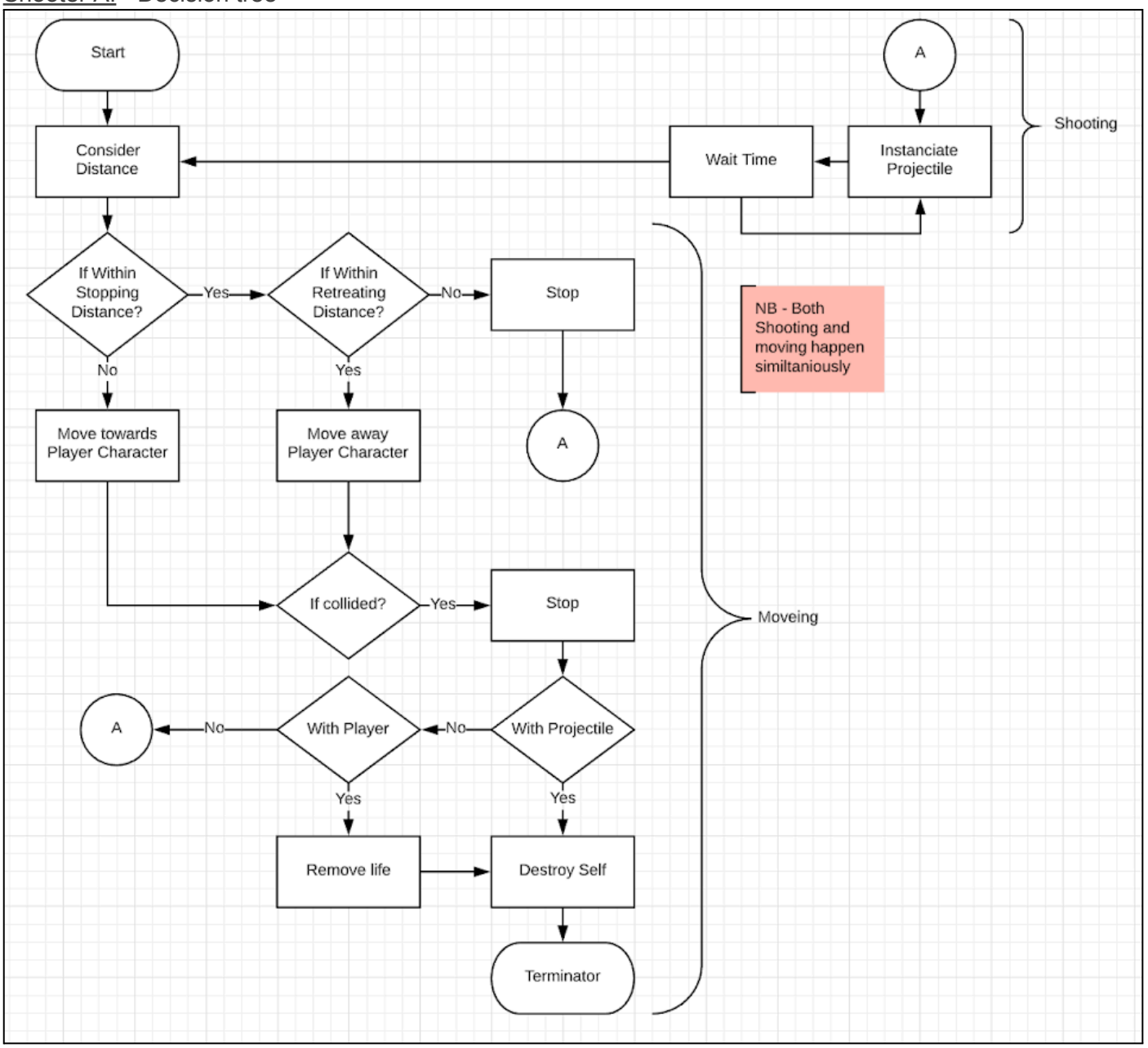
This flowchart attempts to show what steps take place in all parts of the program. The leader board, settings and leave buttons near the start of the program are very linear and therefore work falls in this form, showing clearly what steps need to happen in terms of updates and data manipulation. The game is less clear, after player data is loaded and the game is set up it checks to see if the player is dead, if not it will consider the player inputs and follow a path ending in “A” or “B”. “A” handles movement while “B” handles weapon shooting. In reality, both “A” and “B” can happen on any one iteration but flowcharts struggle when representing simultaneous actions. In addition to the player’s actions, a random enemy is spawned if the spawner conditions are correct, this again is not represented well on the flowchart due to the same issue of simultaneous actions.

In conclusion this flowchart loosely shows the program but misses some events in the program. This, although not a hindrance in development, is important to note for readers so that they do not take this flowchart as a perfect representation of the system. It is a decomposed representation of the solution that is limited by its own nature, a nonlinear system represented by an exclusively linear format.

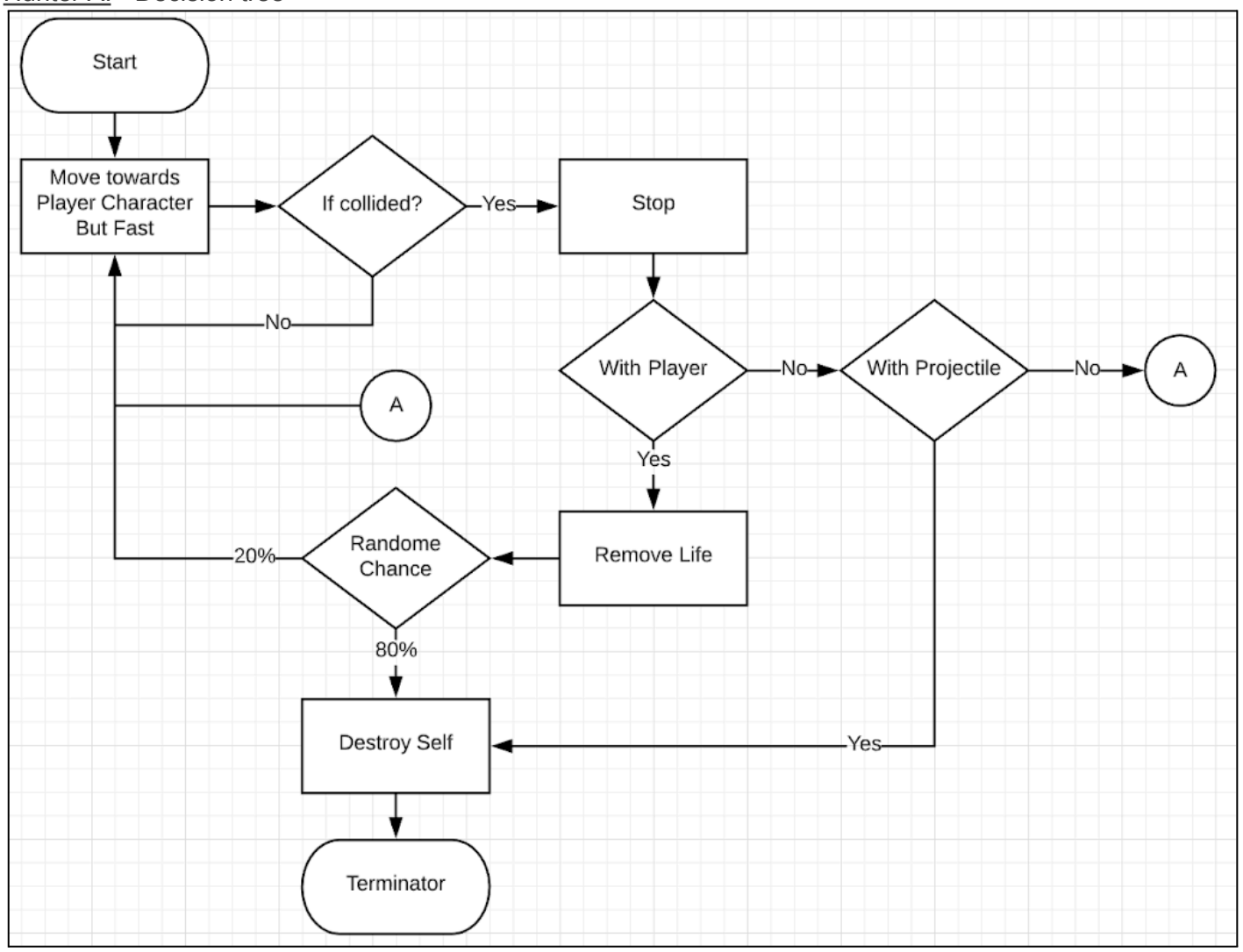
Chaser AI - Decision tree



Shooter AI - Decision tree



Hunter AI - Decision tree



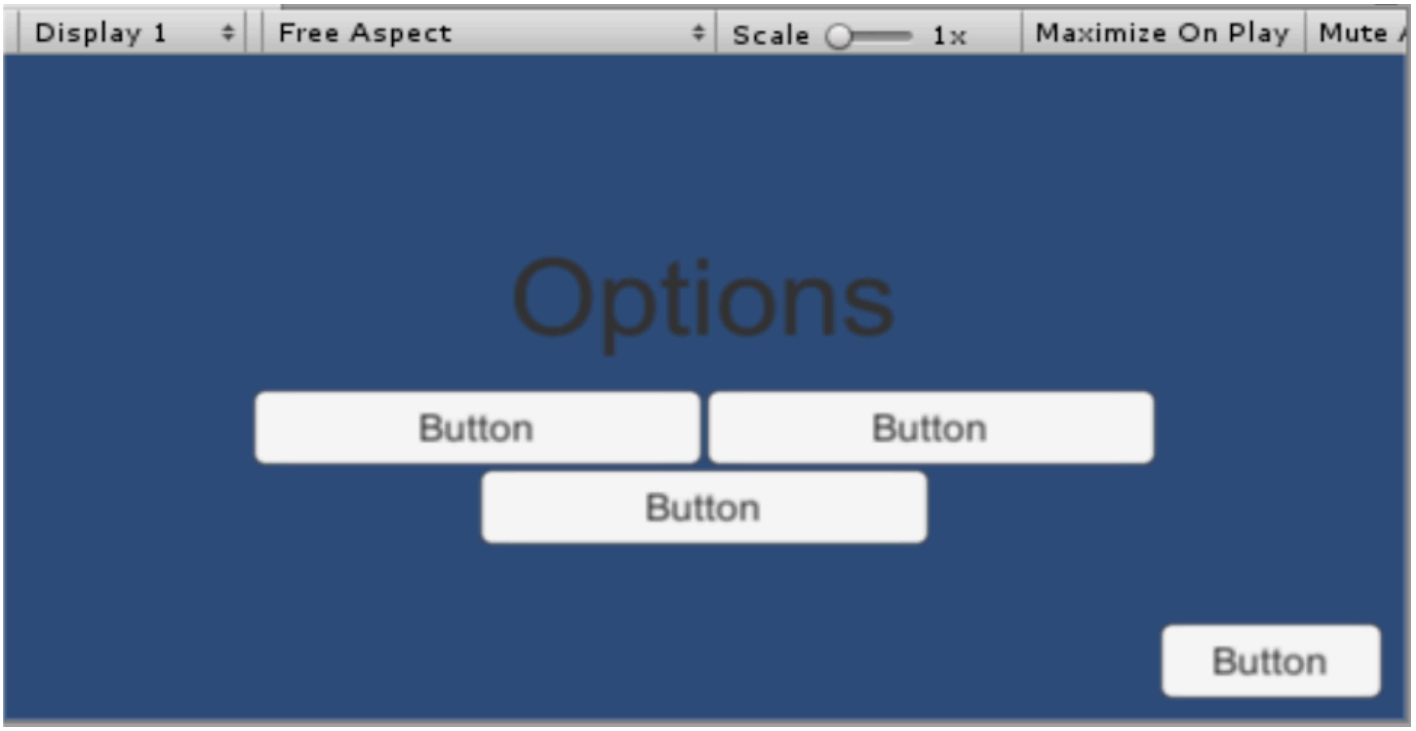
User Interface

Home Screen / Main Menu

A model of the main menu of the game is shown below. First thing you would see. The title sits at the top with three buttons. Top button would say Play and lead to the main game. Middle button would say Settings, leading to a menu for changing various values and game changes such as difficulty. Bottom button would say leaderboard and take you to a list of names and scores. The button in the corner quits the game.
This main page allows the player to go to all parts of the program, it is the transport hub of the program and the place the player almost always ends up at.

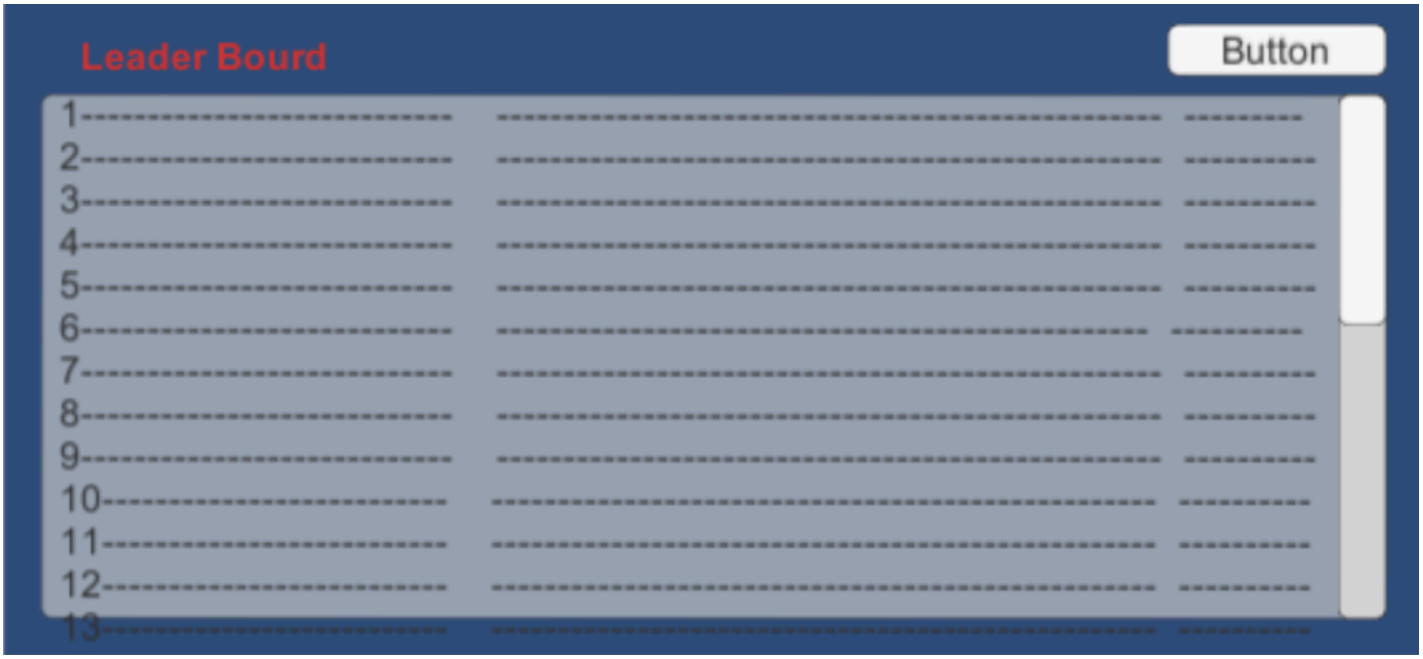
Settings Menu

The options menu will allow the player to change parts of the game, mainly the difficulty, and this will change the speed of enemies, rate of player fire and rate of enemy spawning. This will allow players a more challenging experience to increase the player base.
Scores will be saved and displayed in different locations so that harder play is separate to earlier play

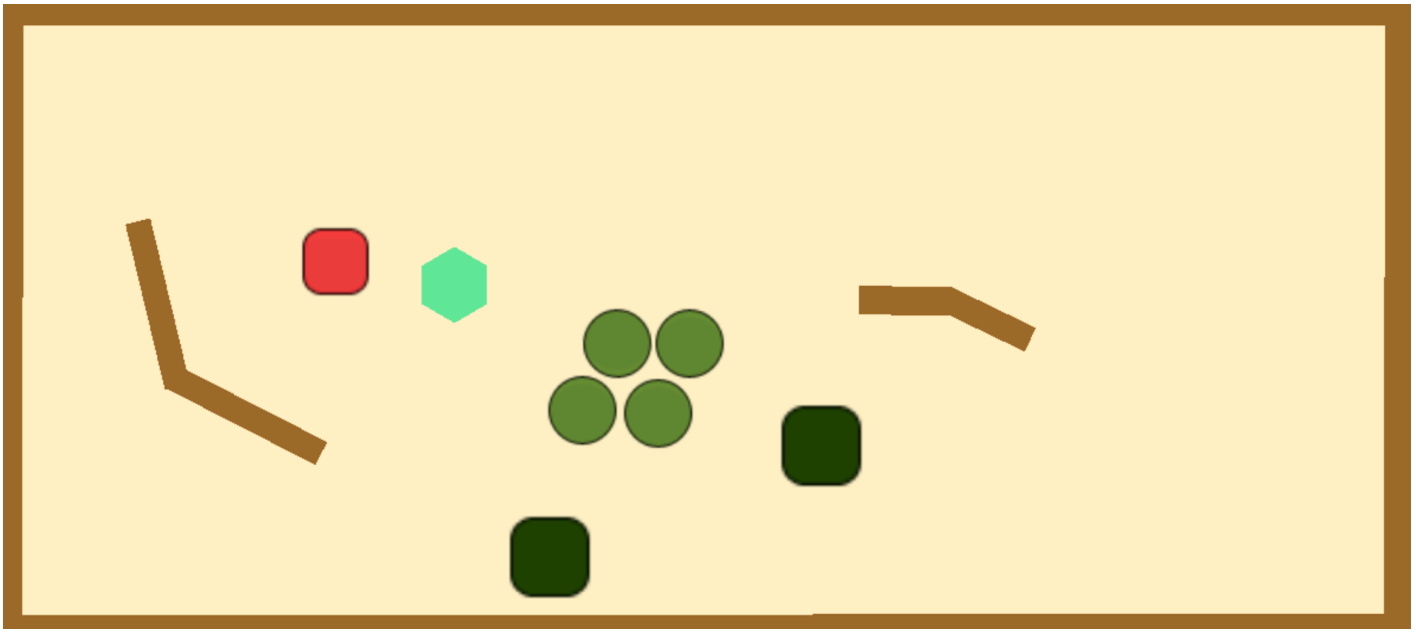


Leaderboard

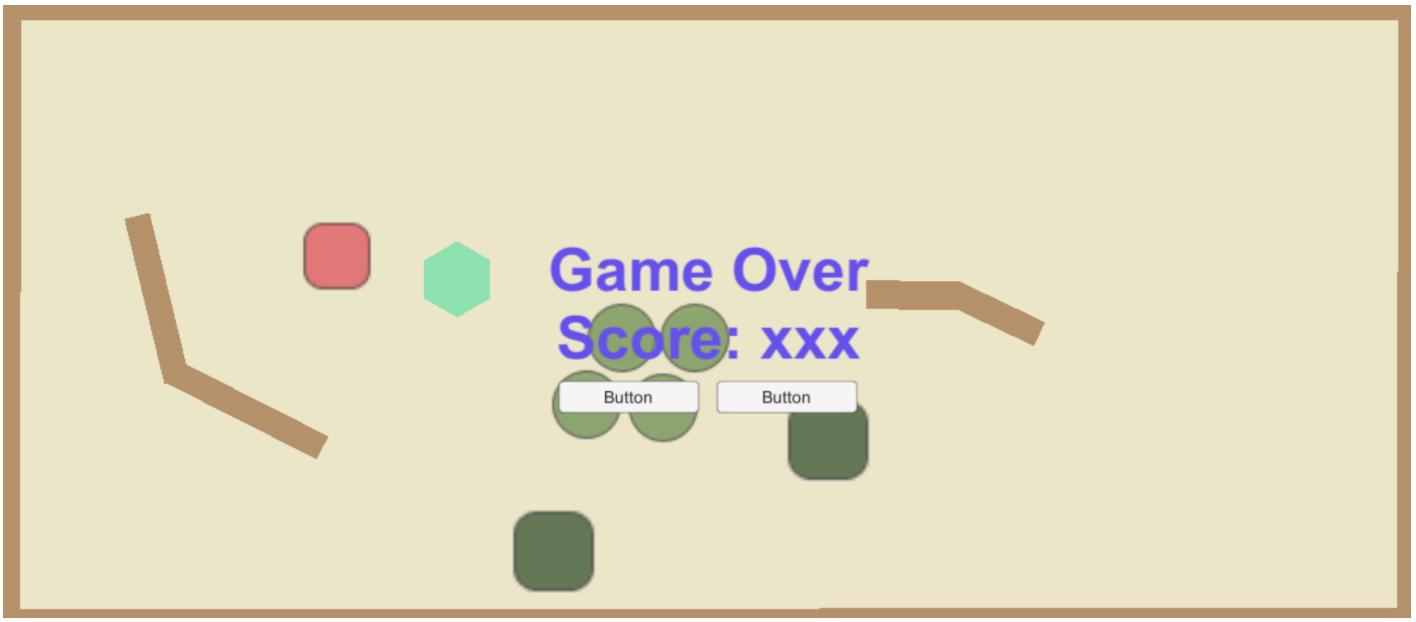
The Leader Board will gain data from a saved file and pass it into the scrolling text box (represented by the dotted lines) where it will be sorted in order of score, allowing players to scroll through all scores.
The button in the top right takes you back to the main menu.
This page allows the user to view a number of previous high scores allowing them to compare themselves to other players and themselves to improve. The use of a scroll bar allows for a greater number of scores to be shown, although the scroll bar may be removed as to only show the best scores.



Main Game
The Red Square is the player
The Green Dots are the Chasers
The Green Squares are the Shooters
The Cyan Hexagon is the Hunter
Brown marks the barrier and the edge of the map.
It is important to note that these are representations and the shapes used (especially the player to be more different) are likely to change.
When the player dies, the game will stop and an overlay will appear
This simplistic style allows for easy to learn entities based purely on colour and shape



One button in the overlay will lead back to the main menu. The other will allow the player to save their score under their username and password to be seen in the leaderboard. On further analysis of this design it is hard to see the text and so all enemies will be deleted when this text is put on screen.



Usability Features

The solution's purpose is to be played, thus it must be easily usable. A list of features that should be known during development so as to allow the solution to be used effectively is set out below.

- Inputs should be clear and have a notable effect - by designing in this way the player can clearly learn the outcome of each input so they can understand the function and its effect. If an input does not have an instant outcome, there should still be an indicator to clarify it has been read as an input. In addition, button objects should be highlighted when hovered over to clarify which button is being pressed.
- The game should attempt to be compatible with the impaired:
    - Visually - the game has a very visual design so levels of blindness will struggle and this cannot be clearly addressed
    - Color - The Colors of the game should be such that a person of any color blindness tripe can still distinguish between the different parts of the solution.
    - Auditory - no sounds are used to indicate things that aren't already indicated visually, therefore no change here.
- The game may lack areas of teaching and direct tutorial, because of this it is important to keep the controls simple and predictable.

Design - Variables, Classes and Data Structures.

Class	Use of class	Variables and Data Struct.	Uses of Var. and DS
Player Character (PC)	Represents the player, centers the movement, controlled with the arrow keys or WASD	speed	Determines the speed of the PC
Weapon	Attached to the PC, the weapon will point towards the mouse cursor and instantiate a projectile object upon left mouse click (shoot)	Type	Determines the type of weapon and the projectile produced
		MC_position	Represents the location of the mouse cursor.
		ShootDelay	Time between shots
Chaser	Enemy that slowly follows the player	speed	Determines the speed of the Chaser
		PC_position	Represents the PCs location
		Difficulty	Amount of score added upon kill
Hunter	Enemy that quickly follows the player	speed	Determines the speed of the Hunter
		PC_position	Represents the PCs location
		Difficulty	Amount of score added upon kill
Shooter	Enemy that moves closer stops then shoots, then retreats if to close	speed	Determines the speed of the Shooter
		PC_position	Represents the PCs location
		Stopping Distance	Distance the Shooter will move to relative to the PC
		Retreat Distance	If PC is closer than this, the shooter will move away from the PC
		Difficulty	Amount of score added upon kill
Projectile	The object that is created by weapons and destroys enemies and player when hit	Speed	Determines the speed of the projectile
		Target	Direction the projectile will move to
Spawner	Object that instantiates enemies	SpawnDelay	Determines the time between spawns
Barrier	Object that bars passage, stops the player or enemies		

Design - Development Testing and Post-Development Testing

Iterative testing will be carried out in 4 steps. The results of the first two will be shown within the Development section at each stage. The last two will be carried out during and after post development testing.

1. White Box Testing - After coding a deconstructed module, the modules functionality will be tested
    - Ensure the module will run as intended
    - Ensure all variables are set appropriately (adjusting things like movement speed) (test shown below)
    - Ensure files save and load correctly
    - Tests Shown Below
2. Black Box Testing - After multiple modules are coded, its inter-modular functionality will be tested to
    - Ensure game success criteria are met
    - Ensure there are no conflicting modules
    - Ensure that all buttons lead to correct locations and all scenes load properly
    - Ensure that all the game's parts flow smoothly
    - NB - due to this form of testing being a repetition of white box testing it will not be recorded in full - only included as an addition if a problem is found
    - After this step a general formal review of the program will be will be carried out
    - Tests Shown Below
3. Post Development Testing (PDT) - Once all modules have been developed and tested the solution will be in its Alpha build. A random sample of users will now test the program. These tests will not have a strict goal apart from them moving through the game. This should find unknown bugs and push it to the limit, as well as providing opinions as to how the game plays and whether it is enjoyable.
    - Collect data on players score - for balancing
    - Collecting data on how different demographics feel about the game - new to experienced Gamers
    - Designing, developing and testing anything missed but the consumer wants
    - These tests won't use any planned test like the white or black box testing as to find gaps in the current testing approaches.
4. Final Test - Now the solution has been put through PDT it will be in its Beta build and can start final testing. Final tests will compare the project directly to the success criteria and ensure that all requirements are met. If a bug is found it will be fixed. If a requirement is not met the Beta build will go back to the Post Development section, solve the problem and then carry out another final test.
    - Ensure the final state of the solution has no known problems and it satisfies the success criteria.
    - Test Against Success Criteria

Validation

The Program will have two form of input:

- Button click / key press - there thing only triggers when the key is pressed and thus will not fire under any other circumstance. Because of this there is no need to validate this input as it's very nature validates itself.
- Text input - Inputting text into a text box takes in a string or characters. Due to this being used to store a username or password there are no specific requirements for this string to contain (it may contain letters, numbers and symbols) and therefore the contents of the string will always be valid. There will be a length requirement that will be validated to check the length of the string.

Design - Development testing points for white and black box testing

Player	
When Up arrow and/or W is pressed the character moves upwards in the space	Testing movement
When Down arrow and/or S is pressed the character moves downwards in the space	
When Left arrow and/or A is pressed the character moves left in the space	
When Right arrow and/or D is pressed the character moves right in the space	
When left mouse click weapon instantiate projectile	Testing weapon function
Weapon tracks the mouse cursor	
All projectiles move forward from instantiation and destroy on collision	
When Player Character touched by chaser loses life	Players life total decreases
When PC touched by hunter loses life	
When PC hit by projectile loses life	
When hits a wall/barrier player cannot move	Testing barriers to ensure restriction work
Enemy	
Enemies are deleted when being hit by projectiles	Checking enemies die as intended as to ensure the game doesn't over stack objects
Chasers and Hunters are deleted when hitting player	
Enemies spawn one at a time, in a “random” order. (note that random still acknowledges that harder enemies spawn later in game)	Enemies spawn at a constant rate and at a level that is good for the player
Enemies damage the player upon attacking it (shot hit or object collision)	Enemies are a threat to the player
Enemies move towards the player as intended (chases and hunters move directly to play until colliding) (shotters move to a point then stop, and move back if to close)	Enemies move as intended
Shooters shoot at player	Shooters are a threat

Display and Inputs	
The health is represented clearly and accurately	Allows player to understand how they are doing in the game, clearly showing rewards and punishments
The score is represented clearly and accurately	
Button click change scene / go to the correct scene	Buttons work correctly
All other inputs should not affect the program	No unintended changes in project
Score	
Score is saved permanently in an external file	Keep track of other users to compare and compeat
Score can easily be displayed with ranking and username	Allow the player to see their, and others, scores

Development

Overview

This is the most non-linear section due to the iterative development model paradigm used throughout. Because if this the layout of each segment to follow is outlined below:-
The program is split into the three sections and then each section split into modules
A section will start with a description of each of its modules being developed and white box tested and then go on to any black box testing point raised and then finish with a general formal review.
A module will start with a small paragraph about that element's function and how it works, the next paragraph will go over the objects involved and the components. Next, any scripts involved will be headed and shown, and commented on so as to show what each part does. Finally, white box testing will take place, going through each test involved in the module, whether it worked and what was done to fix it; This will be followed by illustrations of the working test and, where appropriate, a section that goes through a test solution in more detail. In some cases there may be a part showing a development that happens later in the program, going back to change an element.
After all the modules in all the sections are developed and tested individually, post development testing will take place.

Development - The Game Play

The Player Controller

The first task in creating the game play is the development of the player character.
Firstly, the movement. By creating a 2D sprite in Unity and image can be attached and then place a C# scripts component and a Rigid Body on to it allowing it to translate play inputs into movement. The Rigid Body is a Unity Component that gives that sprite sudi-physics.

The Player Controller is an object of class 2D-Sprite - a 2D-Sprite is a class created by Unity focused on rendering 2D objects, it spawns in with two components; a transform that maps its location in the plain and a sprite renderer that allow for a 2D image to be displayed.
The Player Controller has additional components - a Rigid Body allowing it to experience force, two colliders - one interacts with walls and blocks movement through them, the other triggers collisions between objects.

The C# script “PlayerController”.

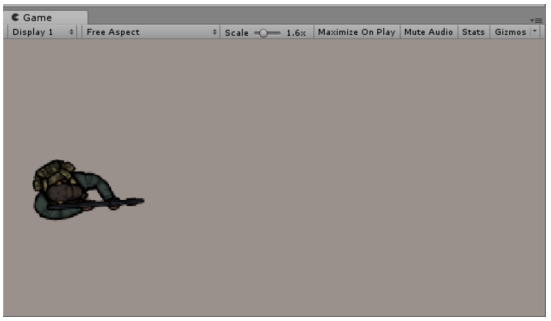
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public float speed = 10; //creates and defines the variable speed. public therefore can change in IDE

    private Rigidbody2D rb; //Creates a variable rb of type 2D Rigid Body
    private Vector2 moveVelocity; //Creates a variable moveVelocity of type 2D Vector

    private void Awake() //Runs at start of program
    {
        rb = GetComponent<Rigidbody2D>(); //define rb to be this components rigid body
    }

    private void Update() //runs every frame
    {
        Vector2 moveInput = new Vector2(Input.GetAxis("Horizontal"), Input.GetAxis("Vertical")); //define moveInput to be equal to any input of vertical or horizontal direction (arrow keys or WASD) - only happens when the appropriate key pressed
        moveVelocity = moveInput.normalized * speed; //normalizes the vector making the value 1 (just direction) then times it by the speed

        rb.MovePosition(rb.position + moveVelocity * Time.fixedDeltaTime); //moves the sprite to its position plus the vector. it's multiplied by Time.fixedDeltaTime so the distance is the same on all processing speeds
    }
}
```

Player controller - White box testing

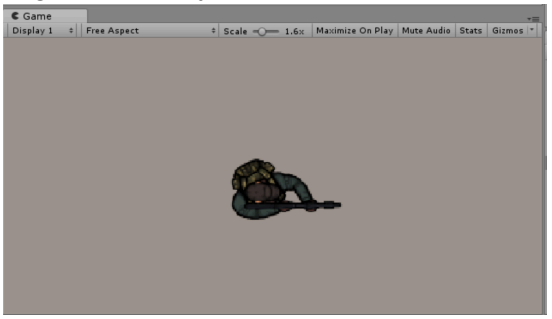
Test	Method	Success?	Solution
When Left arrow and/or A is pressed the character moves left in the space	When the player presses the appropriate key the player moves the appropriate direction, this works this multiple keys with diagonal movement be the same	Success	N/A
When Right arrow and/or D is pressed the character moves right in the space		Success	N/A
When Up arrow and/or W is pressed the character moves upwards in the space		Success	N/A
When Down arrow and/or S is pressed the character moves downwards in the space		Success	N/A

<u>Test illustrations</u>

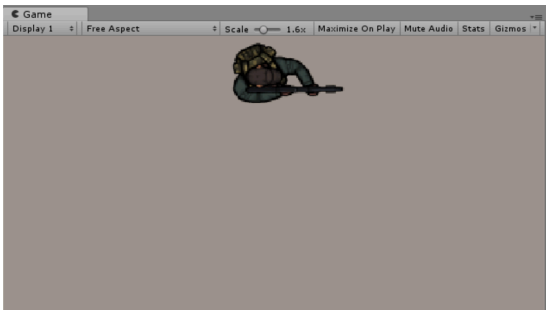
Initial:



Right Arrow key Pressed:



Up Arrow key Pressed:



Down Arrow key Pressed:

Left Arrow key Pressed:

Walls
Walls are non-scripted objects that exist to restrict the movement of sprites.

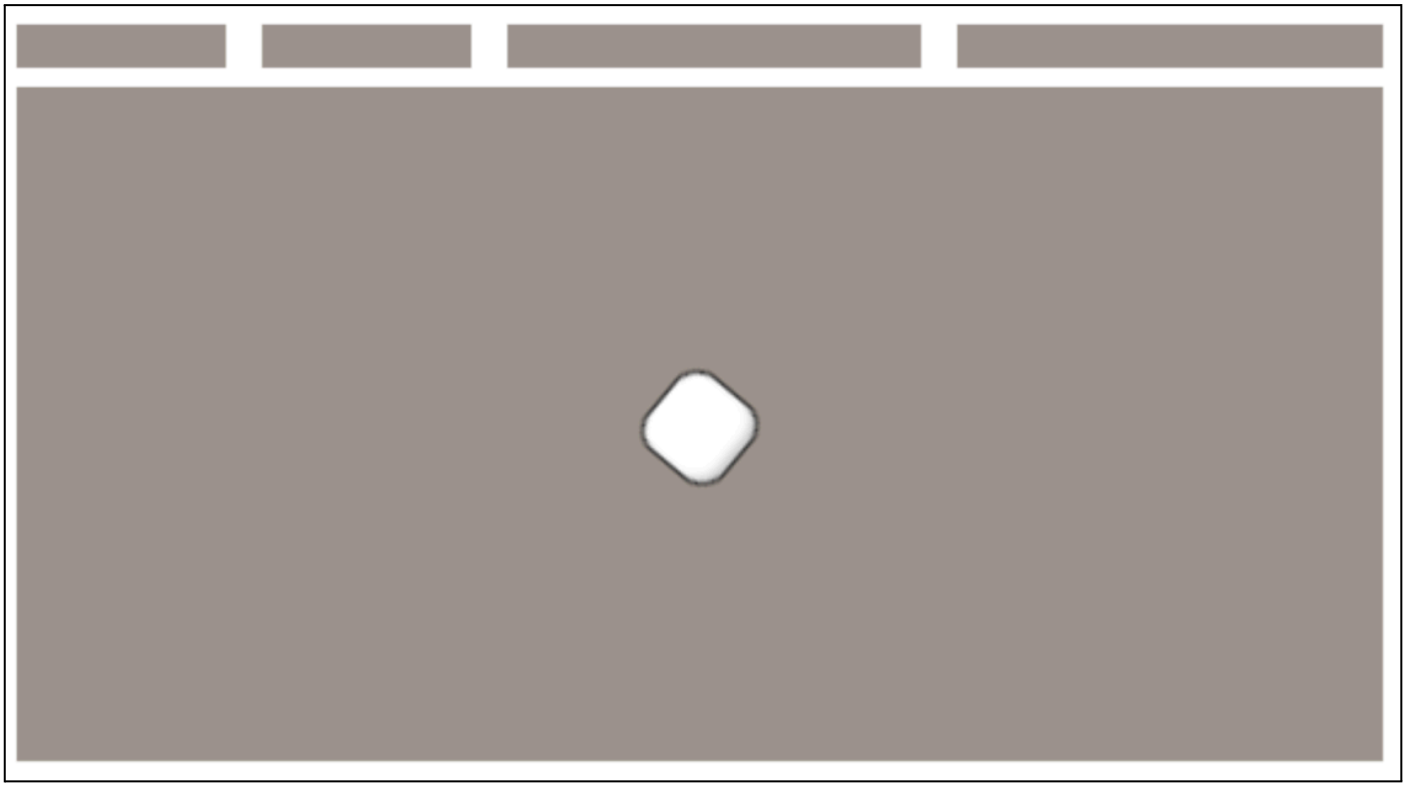
Walls are objects of class 2D-Sprite
They have an additional component - Collider set to stop movement of other objects through it.

No code is included for this module

Walls - White box testing

Test	Method	Success?	Solution
When hits a wall/barrier player cannot move	Testing barriers to ensure restriction work	Success	N/A

Test illustrations



Additional information
Gaps at the top will be for score and health display

The center wall is a fixed impenetrable block to force the player to move around to defend against all enemies, so they can't stand in the middle spinning.

Enemy Chaser and Hunter

Making the enemy follow the player is a task done by all enemies although changes slightly for the shooter. By giving the player the tag “player” the enemies can be caused to fix a target to the player and get the component that parks its location - the two objects have almost identical code but use different game objects

The Enemy Chaser is a prefabricated object of subclass Enemy and superclass 2D-Sprite.
The Enemy subclass has additional components - a Rigid Body, A collider - this just deals with wall physics.
The Enemy Hunter is a prefabricated object of subclass EnemyHunter and superclass Enemy - only difference between sub and super is the code attached and that the hunter has an extra 2D-Sprite object attached to give it a bolder look.

The C# script “EnemyChaser” controls this feature.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class EnemyChaser: MonoBehaviour
{
   public float speed = 5; //variable making speed of the enemy (slower of Chasers / faster for hunters)

   private Transform target; //set a variable to be later defined as the location of the player

   Private void Awake()
   {
      target = GameObject.FindGameObjectWithTag("Player").GetComponent<Transform>(); //defines the game point target as a sprite with the tag “player”’s location (the player)
   }

   private void Update()
   {
      transform.position = Vector2.MoveTowards(transform.position, target.position, speed * Time.deltaTime); //every update the enemy moves towards the target location (the player)
   }
}
```

This code causes the enemy to chase the player around the screen but currently doesn't do anything once hitting the player. The code below should have addressed this by highlighting what part of the collision was not firing correctly.

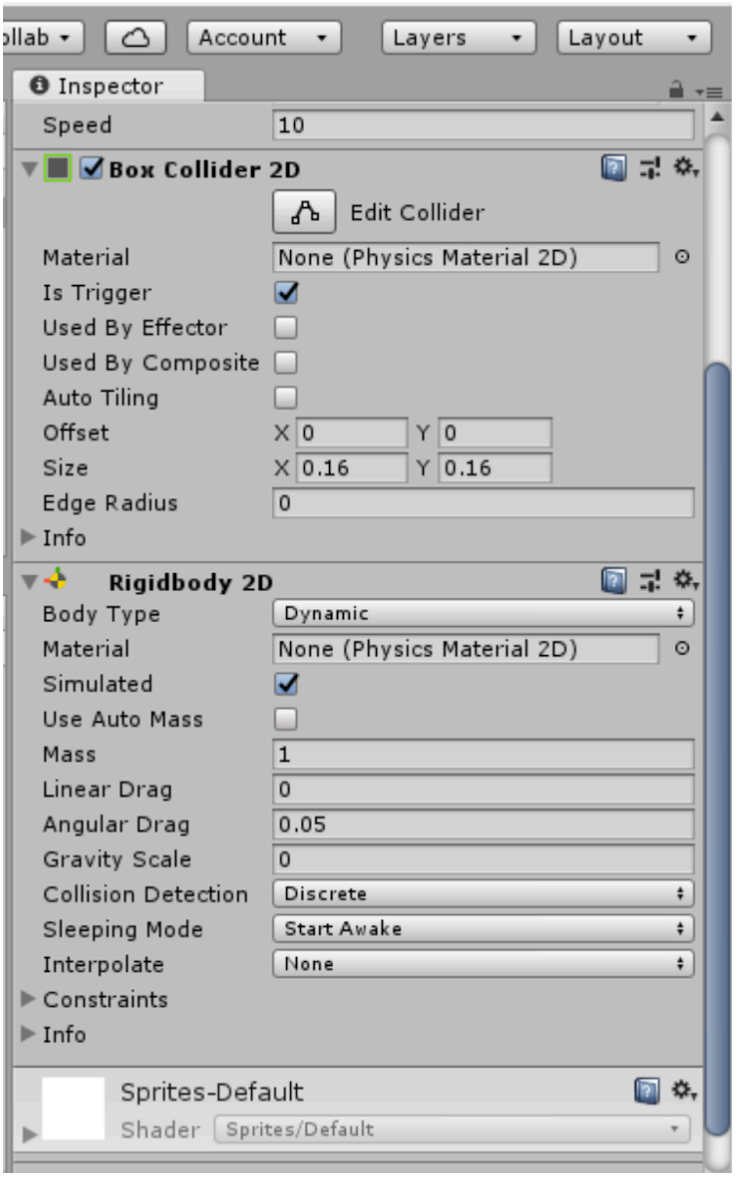
```
   private void OnTriggerEnter2D(Collider2D collision)
   {
      Debug.Log("Collider Player") //Due to the uncertainty of were the error is debug comments were put in that print text into the console

      if (collision.CompareTag("Player"))
      {
         Debug.Log("Collider Player"); //outputs when colliding with a player
      }
      else if (collision.CompareTag("Enemy"))
      {
         Debug.Log("Collider Enemy"); //when colliding with an enemy
      }
   }
```

Solution to a problem in more detail

After writing this code it was found that the collision detection wasn't working.

Shown: https://youtu.be/wQ2iVrrqz4I

After removing the rigid body and collider, then adding a new version it was found that collisions only work in the rigid body is set to kinematic. Then there was the problem that a kinematic object is affected by gravity, easily solved by setting the gravity scale to 0. On the left shows the components involved, the Rigid body 2D and Box Collider 2D, for the player. As you can see the Collider is set to 'Is Trigger', this means instead of stopping it will run the code ***private void OnTriggerEnter2D(Collider2D collision)*** as it is currently printing the messages. By adding ***Destroy(gameObject);*** to destroy the object. This will get it out the way for other objects to interact with the player, anything related to score or health will be added later when looking at the score module.



Shown: https://youtu.be/0UVphFJt2O0

Testing - This section will be tested along with the enemy shooter once that has been developed

Enemy Shooter

The shooter will have a slightly different code for its movement. Still using the same targeting movement but having a line of code that judges the distance between them and stops before getting too close. If the player moves closer to it, it will move backwards until it is greater than the retreating distance

The Enemy Shooter is a prefabricated object of subclass EnemyShooter and superclass Enemy - the only difference is that EnemyShooter has an empty game object attached to the front to give the transform of where to shoot from (instantiate the object at)

The C# script “EnemyShooter” controls this feature.

```
public class EnemyShooter : MonoBehaviour
{
   public float speed = 5; //NB - all numbers seen here and in all code like this are place holders, they will be changed when i bring
                           them all the parts together and make the game more compatible with itself
   public float stoppingDistance = 6.5f; //the f signifies it is a float not an integer
   public float retreatDistance = 5; // the difference between the stopping and retreating distance is the place where the shooter
                                       can sit, not to close not too far away. The shooter doesn't have to be here to shoot or land a hit,
                                       it can shoot anywhere and projectiles should travel indefinitely before collision with wall
   public GameObject Projectile;
   private Transform target;

   void Awake()
   {
      moveTarget = GameObject.FindGameObjectWithTag("Player").transform; //Players location
   }

   void Update ()
   {
      if(Vector2.Distance(transform.position, moveTarget .position) > stoppingDistance) //if too far away
      {
         transform.position = Vector2.MoveTowards(transform.position, moveTarget .position, speed * Time.deltaTime);
           //move closer
      }
      else if(Vector2.Distance(transform.position, moveTarget .position) < stoppingDistance &&
Vector2.Distance(transform.position, moveTarget .position) > retreatDistance) //if close enough and far away enough
      {
         transform.position = this.transform.position; //stay still
      }
      else if(Vector2.Distance(transform.position, moveTarget .position) < retreatDistance) //if too close
      {
         transform.position = Vector2.MoveTowards(transform.position, moveTarget .position, -speed * Time.deltaTime);
           //move away - note -speed
      }
   }
}
```

The next task is to get the shooter to shoot. This will be done by instantiating an object every X frames.

The ShooterProjectile is a game object of subclass projectile and superclass 2D-Sprite
The subclass projectile has additional components - a rigid body, a collider - this collider is only used to trigger collision and delete itself.

Initially this project would be shown from the center of the shooter and travel to a target point (where the player was when the projectile was shot).

The C# script “ShooterProjectile” controls this feature.

```
public class ShooterProjectile : MonoBehaviour
{
   public float speed = 50; //projectile speed - should be the fastest thing in game

   private Transform player;
   private Vector2 target;

   void Start ()
   {
      player = GameObject.FindGameObjectWithTag("Player").transform; //player location
      target = new Vector2(player.position.x, player.position.y); //players location when projectile instantiated
   }

   void Update ()
   {
      transform.position = Vector2.MoveTowards(transform.position, target, speed * Time.deltaTime); //move towards target

      if(transform.position.x == target.x && transform.position.y == target.y) //once at target
      {
          Destroy(gameObject) //destroy self
      }
   }
}
```

but it was found that that would cause the projectile to stop when it reached the target. Instead it was decided to use the same technique that was going to be used for the player weapon. By rotating the shooter to face the player by calculating the angle difference between itself and the player make it face the player, then when the object is instantiated it is facing towards the player so all it has to do is move up (forward) and then will continuously move until it collides with something like the wall.

```
public class ShooterProjectile : MonoBehaviour
{
   public float speed = 50;

   void Update ()
   {
      transform.Translate(Vector2.up * speed * Time.deltaTime); //move forward
   }

   private void OnTriggerEnter2D(Collider2D collision) //if collide with anything
   {
      Destroy(gameObject); //destroy self
      Debug.Log("Hit");
   }
}
```

Although this would also mean needing to rotate the shooter based on the player position and instantiating with the angle of the shooter.

```
public class EnemyShooter : MonoBehaviour
{
   private float timeBtwShots; //variable for time until next shot
   private float startTimeBtwShots = 3; //time delay before start shooting

   public GameObject Projectile; //object that stores Projectile
   public Transform shotPoint;  //the transform of an object that is linked to the EnemyShooter prefab at the edge so that the
                                projectile doesn't hit its firer
   private Transform moveTarget;
   Private Vector3 shootTarget; //the vector location of the player - the target

   private float offset = -90;

   void Awake()
   {
      target = GameObject.FindGameObjectWithTag("Player").transform;

      timeBtwShots = startTimeBtwShots;
   }

   void Update ()
   {
      //Movement controls above

      if(timeBtwShots <= 0)  //when timer = 0
      {
         Instantiate(Projectile, shotPoint.position, transform.rotation); //shoot
         timeBtwShots = startTimeBtwShots; //then reset timer
      }
      else
      {
         timeBtwShots -= Time.deltaTime; //timer ticks down
      }

      shootTarget = GameObject.FindGameObjectWithTag("Player").transform.position; //Vector3 position of player
      Vector3 difference = target - transform.position; //difference between player's location and shooters location
      float rotZ = Mathf.Atan2(difference.y, difference.x) * Mathf.Rad2Deg; //calculate angular difference difference player and
                                                                            shooter

      transform.rotation = Quaternion.Euler(0f, 0f, rotZ + offset); //turn by the angle to face player
   }
}
```

Enemy - White box testing

Test	Method	Success?	Solution
Chasers and Hunters are deleted when hitting player	Allowing the enemies to hit the player	Initially no, the collision were not detecting	Ensuring the collision module was set to ‘As trigger’ so it ran the collision code when there was a collision
	Second test	Success	N/A
Enemies move towards the player as intended (chases and hunters move directly to play until colliding) (shooters move to a point then stop, and move back if to close)	Moving the player and seeing if the enemies chase the player as intended	Success	N/A
Shooters shoot at player	Allowing the shooters to shoot	Initially no, the angles were not correct	By changing the models of both the shots and the shooter it was found that the problem was with the rotation of the shooter. Adding in an integer value to offset the angle ensured it shot strategies.
	Second test	No, the angle was still off	Due to the variable being public, it changed when the program started due to an input in the script component in the IDE. the variable was made private
	Third test	Success	N/A

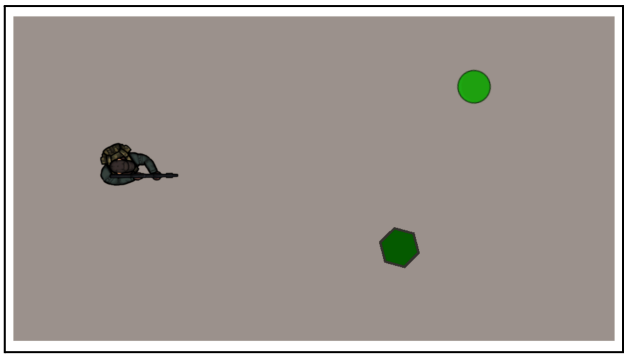
Test Video - Enemy Shooting
The rotation of the shooter was a problem initially as the sprite used was not orientated properly. After this was changed the shooter was still oriented wrong.

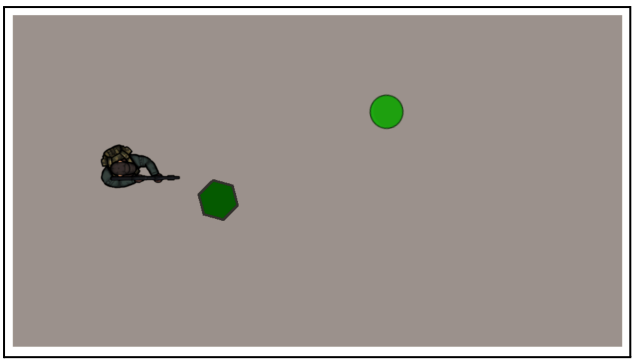
Video Test: https://youtu.be/gnIMiA0CHYk

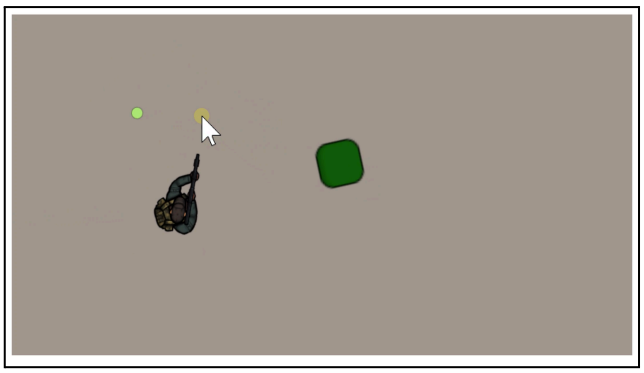
After showing setting the offset to private the offset worked well.
This feature was then added to all other tracking prites (player character and other enemies), the player character uses the mouse cursor instead of the player.

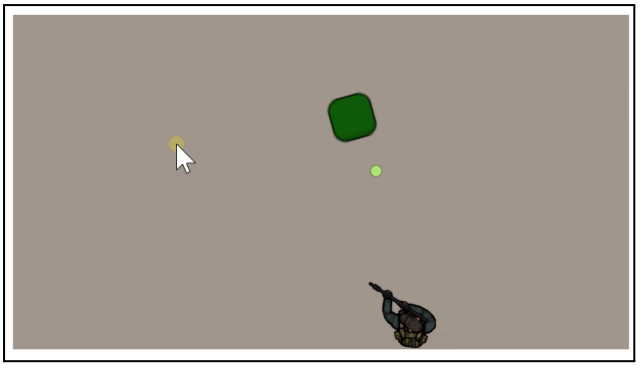
Test illustration - Enemy Movement

Three shots showing the movement of the enemy units towards the player. note how the hunter (darker green hexagon) moves around twice as fast.





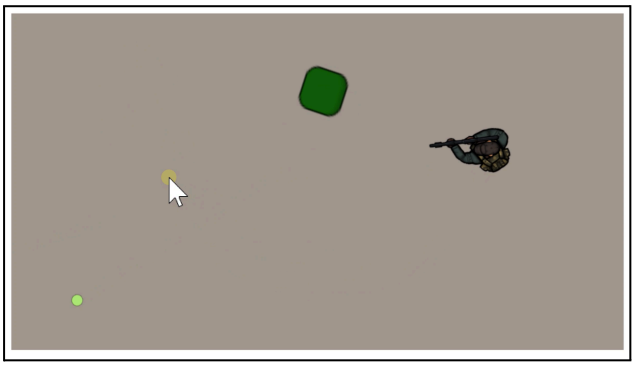




These three images try to show the movement and facing of the player and enemy.
It can be seen that the player will always face the mouse cursor and the enemy will always face the play.
In the last one it can be seen how the enemy has shown a projectile (small lime green dot) and the player has moved round, was facing there when the shot was made but then has turned.
The link below is to a video that shows this test in its entirety.
https://www.youtube.com/watch?v=5wKtoZCinQg

Spawners

The spawner works on a tick down system. Every update a set value is reduced by an amount, when the timer reaches 0 an enemy is spawned and the timer is reset. To increase difficulty over time the timer is reset to a smaller value than before so the spawning interval is decreased.



The spawner is an object of class 2D-Sprite

The C# script “ShooterProjectile” controls this feature.

```
public class Spawner : MonoBehaviour {
    public GameObject enemy1; //Game objects will be given the prefabricated objects of the enemies.
    public GameObject enemy2;  //enemy1 = chaser, enemy2 = shooter, enemy3 = hunter
    public GameObject enemy3;
    private int count = 0; //count the number of times enemies have spawned to allow harder enemies to spawn later in the game
                           when count is high
    private int ShooterStart = 10; //number of spawns shooters will start
    private int HunterStart = 50;  //number of spawns hunters will start
    private int randNum; //placeholder for a randomly generated number
    public float timeBtwSpawn = 0; //time until next spawn - counts down through the program
    public float startTimeBtwSpawn = 5; //the number timeBtwSpawns is set to after a spawn
    public float timeChange = 0.01f; //the amount startTimeBtwSpawn changes every spawn (decrease)

    private void Spawn(int phase) //function that will spawn a random enemy
    {
        phase++; //the “phase” determines what enemies can spawn - phase++ is faze +1 to make the maths work
        randNum = Random.Range(1, phase);
        if (randNum == 1) //spawns just enemy1
        {
            Instantiate(enemy1, transform.position, Quaternion.identity);
        }
        else if (randNum == 2) //spawns enemy1 or enemy2
        {
            Instantiate(enemy2, transform.position, Quaternion.identity);
        }
        else if (randNum == 3) //spawns any enemy
        {
            Instantiate(enemy3, transform.position, Quaternion.identity);
        }
    }
```

//Continues below

```
void Update ()
  {
     if(timeBtwSpawn <= 0) //if the count down to spawn = 0
     {
        count = count + 1; //count the spawn

        timeBtwSpawn = startTimeBtwSpawn; //reset the clock
        startTimeBtwSpawn = startTimeBtwSpawn - timeChange; //decrease the max on the clock

        if(count <= ShooterStart) //start of the game
        {
           Spawn(1); //phase 1
        }
        else if(count > ShooterStart && count <= HunterStart) //mid game
        {
           Spawn(2); //phase 2
        }
        else if(count > HunterStart) //late game
        {
           Spawn(3); //phase 3
        }
     }
     else //if time doesn't = 0
     {
        timeBtwSpawn -= Time.deltaTime; //timer ticks down
     }
  }
}
```

Spawners - White box testing

Test	Method	Success?	Solution
Enemies spawn one at a time, in a "random" order. (note that random still acknowledges that harder enemies spawn later in game)	Allowing the enemies to spawn	Initially no, not spawning in the correct phase	Time between phases was too short so between spawn 1 and 2 it was moving through all phases. Move count form every update to every spawn.
	Second test	Success	N/A

Player Shooting, Enemy Kills and Score

This may seem like a lot of steps, but to build and test the weapon the enemies must be able to die, this means implementing projectile collision. Due to this feature being fundamentally linked with the score, counting system for the score will be added alongside this although the actual saving loading and score boards part of the score will be developed in another section

Editing the player controller code to implement score and health.

The C# script “PlayerController”

```
public int Lives = 3;
public int score = 0;

private void OnTriggerEnter2D(Collider2D collision) //if collision reduce the health by 1
{
    if (collision.CompareTag("Enemy")) //multiple tags doing same thing here will do different things later
    {
        Lives--; //reduces lives by one
    }
    if (collision.CompareTag("Projectile"))
    {
        Lives--;
    }
    if (collision.CompareTag("EnemyProjectile"))
    {
        Lives--;
    }
}

// ← code from before

private void Update() //new bit
{
    if(Lives <= 0) //for now the player disables itself when its health = 0 but will change when we have the menu structures
    {
        gameObject.SetActive(false);
    }

//rest of code continues like before
```

Moving to all enemy codes add this to the start of the code to link the variables in player controller and the enemies.

The C# script “ShooterProjectile”

```
GameObject player;
PlayerController PlayerController;

void Awake()
{
    player = GameObject.FindGameObjectWithTag("Player"); //code to link with the PlayerController code so we can eddite
                                                         variables
    target = player.GetComponent<Transform>();
    PlayerController = player.GetComponent<PlayerController >();
}
```

And this at the collision detection part to change the score and destroy self - these will destroy self in final version

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Player"))
    {
        Destroy(gameObject);
    }
    if (collision.CompareTag("Projectile"))
    {
        Destroy(gameObject);
        PlayerController .score = PlayerController .score++; //increase score by one
    }
    if (collision.CompareTag("EnemyProjectile"))
    {
        Destroy(gameObject);
    }
}
```

All this code worked although a problem was found. The player and enemy would collide 3 times before the object deletes itself. One collider can be turned off. It is reduced to 2. Given that this would be the simplest solution, the value of a collision will be changed so that every collision will reduce by half resulting in a game collision being equal to two half real collisions resulting in it working as intended.

So

```
Lives = Lives - 0.5f;
```

And

```
PlayerController .score = PlayerController .score + 0.5f;
```

It was later found that, as only one of the players and enemies needed to trigger the collision, then by scaling up all objects in the program and changing the speed the double collision problem would not happen for slower objects. The enemies will use a change of 1 but projectiles will use a change of 0.5 due to the double collision problem still occuring.

The weapon

Moving on to the weapon. Originally this was going to be carried out by rotating a separate object on the player. However, due to this not being needed, using just the player will result in less objects, better processing and less separate code. The player will rotate to face the mouse cursor by calculating the angle between it and the cursor and then rotating that difference. The mouse click will then instantiate a projectile at the end of the weapon and that will kill enemies.

The C# script “PlayerController”

```
faceTarget = Camera.main.ScreenToWorldPoint(Input.mousePosition); //This Vector3 definition is added at the start - face position is the mouse position
Vector3 difference = faceTarget - transform.position; //new Vector3 is difference between
float rotZ = Mathf.Atan2(difference.y, difference.x) * Mathf.Rad2Deg; //new float is how much to rotate

transform.rotation = Quaternion.Euler(0f, 0f, rotZ + offset); //rotation code, 0 in x and y, rotZ in Z

if (Input.GetMouseButtonDown(0)) //if left mouse click
{
    Instantiate(projectile, shootPoint.position, transform.rotation); //instantiate object
}
```

And finally the projectile code - The play projectile is also a game object of class Projectile

```
public class PlayerProjectile : MonoBehaviour
{
    public float speed = 50; //speed of projectile

    void Update()
    {
        transform.Translate(Vector2.up * speed * Time.deltaTime);
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            Destroy(gameObject);
            Debug.Log("Hit Player");
        }
        else if (collision.CompareTag("Enemy"))
        {
            Destroy(gameObject);
            Debug.Log("Hit Enemy");
        }
        else if (collision.CompareTag("Wall"))
        {
            Destroy(gameObject);
            Debug.Log("Hit Wall");
        }
    }
}
```

Weapons system and scoring - White box testing

Test	Method	Success?	Solution
When left mouse click weapon instantiate projectile	Left click and see if projectile instanciates	Success	N/A
Weapon tracks the mouse cursor	Move the mouse and see if player rotates	Success	N/A
All projectiles move forward from instantiation and destroy on collision	Instantiate the objects and see if it moves forward. Repeat this task until it collides with all the objects.	Success	N/A
When Player Character touched by chaser loses life	Player touch chaser	Success	N/A
When PC touched by hunter loses life	Player touch Hunter	Success	N/A
When PC hit by projectile loses life	Player Hit with all projectiles	Success	N/A

Test Evidence

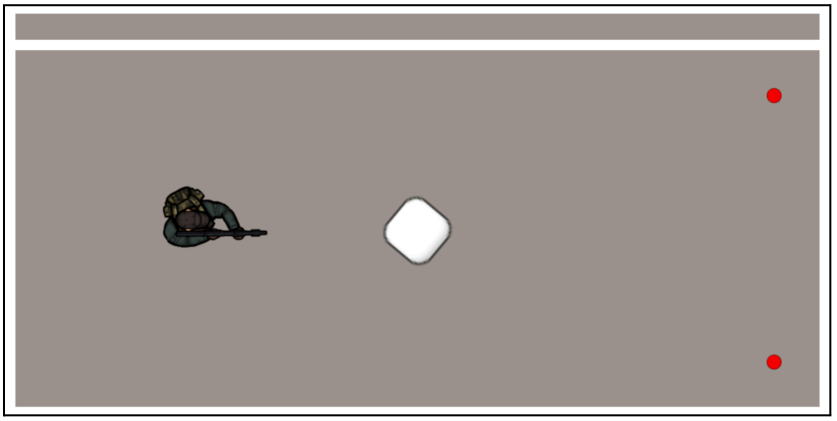
This module's tests are best shown in a video. Here the speed of the enemies was reduced to 0 to make testing easier. The score recorder will be moved later.
https://www.youtube.com/watch?v=4fLFPDFpDEE&feature=youtu.be

Later development

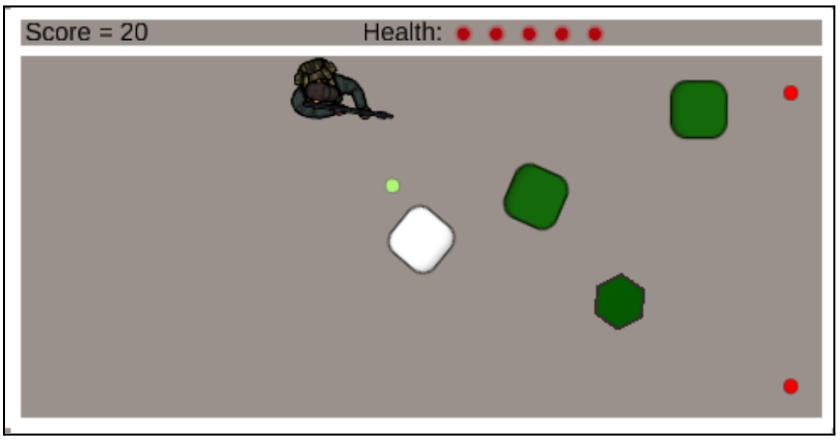
After noting that play projectile and enemy projectile worked identically, the objects remain separate with different tags although they both have the same code attached - 'Projectile'

The Game Display

The last part to develop before the game play is tested together is the display. Currently there is a box format. The white areas are walls, these have no code but are just made with collider components with a trigger not checked, including the white in the center. The small red nodes are the spawners although they will not be visible to the player in the real thing. The player is the humanoid modal to the left of the center block. Finally there is a band at the top. This is where the player will have their health and their score shown.



At the top the Score was added on the left and health in center right, both at the top. The score is represented textually though integer in a text box and the health shown by showing hiding images. All code for this is done in the Player Controller. Two functions were written, ScoreUpdate() and HealthUpdate(). These functions shown below change their respective displays and are called after any of the values are changed. For example, in Player Controller, when the player is hit, Life is reduced by 1 and HealthUpdate is called.



All UI objects are classes pre-built be Unity with little to no component changing - includes Canvas, Panels, Buttons, Texts, images

The C# script “PlayerController”

```
private TextMeshProUGUI scoreText; //set to the score text box

public Image Lives1; //set to on the read dots - the life points
public Image Lives2; //Lives1 set to the first life point, Lives5 set to the last life point
public Image Lives3;
public Image Lives4;
public Image Lives5;

public void UpdateScore()
{
   scoreText.text = "Score = " + score; //set the text to “Score: whatever the score is”
}

public void UpdateLife()
{
   if (Lives <= 4) //objects representing lives are hidden
   {
      Lives5.enabled = false; //hide the life point
   }
   if (Lives <= 3)
   { … } //same code
   if (Lives <= 2)
   { … }
   if (Lives <= 1)
   { … }
   if (Lives <= 0)
   {
      Lives1.enabled = false; //hide the life point
      gameObject.SetActive(false); //delete the play
   }
}
```

The game display - White box testing

Test	Method	Success?	Solution
Both the health are represented clearly and accurately	Take damage and see if the health decrease	There was a problem if i took two lots of damage simultaneously as originally the if statements were absolute ***if(Health == 5) { }***	Change the if statements to less then or equal to ***if (Lives <= 4) { }***
		Success	N/A
Both the score are represented clearly and accurately	Kill enemies and see if the score change	Success	N/A

Development - The Game Play - Formal review

At this stage in the game development, all modules of the game play section have been developed and put in place. No problems were found in Black box testing. Things to still develop are the menu systems; this will be the next steps; as well as the scoring along with the score table, saving and loading. In addition to this the variables, such as timing, speed and implementing delays, balance the game. Currently the program spawns at ridiculous rates and there is no limitation to shooting. This will be addressed after all program code is developed in the post development phase.

Development - The Main and Options Menu

The Menu system is highly dependent on the tools made available by Unity and the Asset Library. The Menus are built from the use of a canvas. Buttons (as well as text and images) are placed onto the canvas, they run code when pressed. When a button is pressed the scene changes. A scene is the collection of objects that make up a part of the game, the main menu is a scene consisting of button and text objects, the game is another scene consisting of objects like the player and enemies. When a scene is loaded all objects of the previous scene are removed and all objects of the new scene are loaded on to the page. The Play button loads the main game scene. The options button loads the options scene, where the player can change the difficulty of the game. The scoreboard button loads the scoreboard scene. The quit button kills the program. Buttons have a visual change when hovering and pressing them, as shown by the options button here.

The code is linked to a blank object named “UI” in each scene and the buttons use their respective function for the script attached to it. Code for Options buttons.

The C# script “btnScript”

```
public class btnScript : MonoBehaviour {

   public int difficulty = 1;  //value to set the difficulty

   public void Easy() //when button pressed this function is called
   {
      difficulty = 1; //the value is changed
      Debug.Log("Easy Button Pressed");
   }

   public void Medium()
   {
      difficulty = 2;
      Debug.Log("Medium Button Pressed");
   }

   public void Hard()
   {
      difficulty = 3;
      Debug.Log("Hard Button Pressed");
   }

   public void Back()
   {
      SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex - 1); //changes the scene back to the main menu
      Debug.Log("Back Button Pressed");
   }

   public void Play()
   {
      SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 3);
      Debug.Log("Play Button Pressed");
   }

   public void Options()
   {
      SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
      Debug.Log("Options Button Pressed");
   }

   public void Score()
   {
      SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 2);
      Debug.Log("Score Button Pressed");
   }

   public void Quit()
   {
      Application.Quit();
      Debug.Log("Quit Button Pressed");
   }
}
```

The Influence of the Options

The options dictate the difficulty of the game. The higher the difficulty the higher the spawn rate with harder enemies spawning faster. Scores will also be worked into a new table to clarify the harder score acquisition.

Anywhere where speed or time are used the value will be multiplied by the difficulty. Speed of projectile, speed of play and enemies as well as speed of spawning. Medium is the rate the game is intended to be played, easy is half speed for everything and hard doubles the speed for everything. This means that the difficulty is based mainly on reaction time and control.

The C# script “PlayerController” - Movement and weapon reloading

```
rb.MovePosition(rb.position + moveVelocity * difficulty * Time.fixedDeltaTime);

if (reloading) //if you are reloading
    {
      if (tempTime <= 0) //if reloading timer = 0
      { // reloading code}
      else //if time doesn’t = 0
      {
         tempTime -= Time.deltaTime * (difficulty); //timer ticks down
      }
    }
```

There will be a separate scoreboard for each difficulty allowing for an accurate comparison for the game played.

The game display - White box testing

Test	Method	Success?	Solution
Button click change scene / go to the correct scene	Click all the buttons, and see if they work correctly	Success	N/A
All other inputs should not affect the program	Press all non defined inputs (all inputs that shouldn’t do anything) at all stages of the program and investigate the output	Success	N/A

Development - Black Box Testing - The Menu and The Game

When implementing the menu system it was found that when the game ends you are sent back to the main menu, but if you were to go back into the game many items wouldn't be initialised. All objects that exist under the object ---Game--- (an object used mainly to organise the objects but is disabled at the end of the game to remove all game objects form the UI objects) would still be disbaled. The simple solution is having a line of code that when ***Awake ()*** is called sets all the objects to the state they could be in at the start of the game.

This code "Enable_Game" is attached to an empty game object and runs when the scene MainGame starts

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Enable_Game : MonoBehaviour {

   public GameObject game;

   void Awake ()
   {
      game.SetActive(true);
   }
}
```

Development - The Menus - Formal review

At this stage of the program there is a functioning game with a menu system. From the main menu you can go to the main game and play it, after which you are sent back to the main menu. From the main menu you can also go to options and change the difficulty. The button labeled Scoreboard leads to a blank scene.
I will now develop the scoreboard.

Development - The Scoreboard

On analysis of existing solutions, the initial plan was to save variables to a text file and bubble sort although it was found through a video that it would oversimplify the saving and the bubble sort was not necessary. The revised method consists of creating a structure (structure - a collection of variables collected like a unique data type) that will contain the players name (sting) and score (int). This will be stored in a list within an object. This object's data will be added to and stored in a text file - .Json file specifically. The sorting can happen when adding data. Only the top values are needed so if the new value is smaller than the top 10 it is discarded. If it is higher it is inserted into the appropriate place and the last value is removed. The result is a list of scores that undergo an insertion sort when a new value is added.
During development we found that some of the design variables could be removed from the players score. Instead of showing the number of games a player has played, we can just show their scores separately, such that one player can hold the best and second best score. In addition, we concluded that as no personal data is being stored the requirement of a password is unnecessary. This leaves us with just a username and score to save. We also set it so their username can only be 3 characters so add to the retro arcade feel of the scoreboard.

The C# script “ScoreboardEntryData”

```
[Serializable]  //allows data to be stored to a file

public struct ScoreboardEntryData  //crease a struct (data type of data types)
{
   public string entryName;  //holds 3 character name
   public int entryScore;  //holds score
}
```

The C# script “ScoreboardSaveData”

```
[Serializable]

public class ScoreboardSaveData  //an object that will store a list which will store the scores
{
   public List<ScoreboardEntryData> highscores = new List<ScoreboardEntryData>();  //creates a list of
‘ScoreboardEntryData’s
}
```

The C# script “ScoreboardEntryUI”

```
public class ScoreboardEntryUI : MonoBehaviour
{
   [SerializeField] private TextMeshProUGUI entryNameText = null; //link the text object to past into
   [SerializeField] private TextMeshProUGUI entryScoreText = null; //link the text object to past into

   public void Initialise(ScoreboardEntryData ScoreboardEntryData) //assign the variables
   {
      entryNameText.text = ScoreboardEntryData.entryName;
      entryScoreText.text = ScoreboardEntryData.entryScore.ToString();
   }
}
```

The C# script “Scoreboard”

```
public class Scoreboard : MonoBehaviour
{
    [SerializeField] private int maxScoreboardEntries = 8; //max number of scores in list
    [SerializeField] private Transform highscoresHolderTranform; //location of where the scores should be instantiated
    [SerializeField] private GameObject scoreboardEntryObject; //object to instantiate objects into

    private string SavePath => $"{Application.persistentDataPath}/Highscores.json"; //Save location of the text file

    public bool OnHighscoreTable; //the code used to save data to the list is used in multiple scenes, therefore their score table can only be updated if you are on the high score table, therefore a public bool is used to show this.

    private void Start() //runs when the scoreboard scene is opened
    {
        ScoreboardSaveData savedScores = GetSavedScores();

        UpdateUI(savedScores);
    }

    public void AddEntry(ScoreboardEntryData scoreboardEntryData)
    {
        ScoreboardSaveData savedScores = GetSavedScores();

        bool scoreAdded = false; // does the item fit in the list

        for (int i = 0; i < savedScores.highscores.Count; i++)  //repeat for the length of the list
        {
            if (scoreboardEntryData.entryScore > savedScores.highscores[i].entryScore) //if bigger any value - starting from the first one
            {
                savedScores.highscores.Insert(i, scoreboardEntryData); //insert it in the list at the right place
                scoreAdded = true;  //not that the value is in the list
                Break; //stop comparing values
            }
        }

        if (!scoreAdded && savedScores.highscores.Count < maxScoreboardEntries) //if there is less then the max values and the value is not already added, add value to the end of the list
        {
            savedScores.highscores.Add(scoreboardEntryData);
        }

        if (savedScores.highscores.Count > maxScoreboardEntries) //if there are too many items in the list, remove the extra items
        {
            savedScores.highscores.RemoveRange(maxScoreboardEntries, savedScores.highscores.Count - maxScoreboardEntries);
        }

        SaveScores(savedScores); //save the new list to the text file

        if(OnHighscoreTable) //if on the high score screen
        {
            UpdateUI(savedScores); //load and update the table
        }
    }


    private void UpdateUI(ScoreboardSaveData savedScores)
    {
        foreach (Transform child in highscoresHolderTranform)
```

```
        {
            Destroy(child.gameObject); //remove the existing values
        }

        foreach (ScoreboardEntryData highscore in savedScores.highscores)
        {
            Instantiate(scoreboardEntryObject, highscoresHolderTranform).GetComponent<ScoreboardEntryUI>().
Initialise(highscore); //instantiate objects to values to the list
        }
    }

    private ScoreboardSaveData GetSavedScores() //to save data
    {
        if (!File.Exists(SavePath)) //if there is no existing table
        {
            File.Create(SavePath).Dispose(); //create a file
            return new ScoreboardSaveData(); //and add return the empty list
        }

        using (StreamReader stream = new StreamReader(SavePath)) //reading from the list
        {
            string json = stream.ReadToEnd(); //read the values into a string 'json'

            return JsonUtility.FromJson<ScoreboardSaveData>(json); //return values from the list
        }
    }

    private void SaveScores(ScoreboardSaveData scoreboardSaveData)
    {
        using (StreamWriter stream = new StreamWriter(SavePath)) //Writing a sting
        {
            string json = JsonUtility.ToJson(scoreboardSaveData, true); //link the values to write into a string json
            stream.Write(json); //write values into string
        }
    }
}
```

The C# script “btnScript” used by the button that appears once the game ends

```
public void SaveScore()
  {
    ScoreDesctiption.enabled = false; //changing visual display

    string value = InputName.text; //take value of the input box

    if (value.Length != 3) //validation - if not 3 character
    {
      ScoreNotValid.enabled = true; //show error menu
    }
    else
    {
      SaveScore_btn.enabled = false; //disable save button

      NewScore.entryScore = PlayerController.score; //save value for score
      NewScore.entryName = InputName.text.ToUpper(); //save username with uppercase

      Scoreboard.AddEntry(NewScore); //save the new score using function from code ‘Scoreboard’

      SceneManager.LoadScene(0); //to main menu
    }
  }
```

The Scoreboard - White box testing

Test	Method	Success?	Solution
Score is saved permanently in an external file	Attempt to save a number of scores. Close the program and then open, running the program and seeing if the scores are saved.	All scores were saved although not all in the correct format with the username. Usernames didn't have to be 3 characters, one even had no characters, and many were lowercase.	Have a validation to catch the number of characters. ***if (value.Length != 3)*** And set the case of the user name to upper. ***NewScore.entryName = InputName.text.ToUpper();***
		Success	N/A
Score can easily be displayed with ranking and username	Go to the scoreboard menu to see if all highscores are loaded.	Success	N/A

Graphical Change

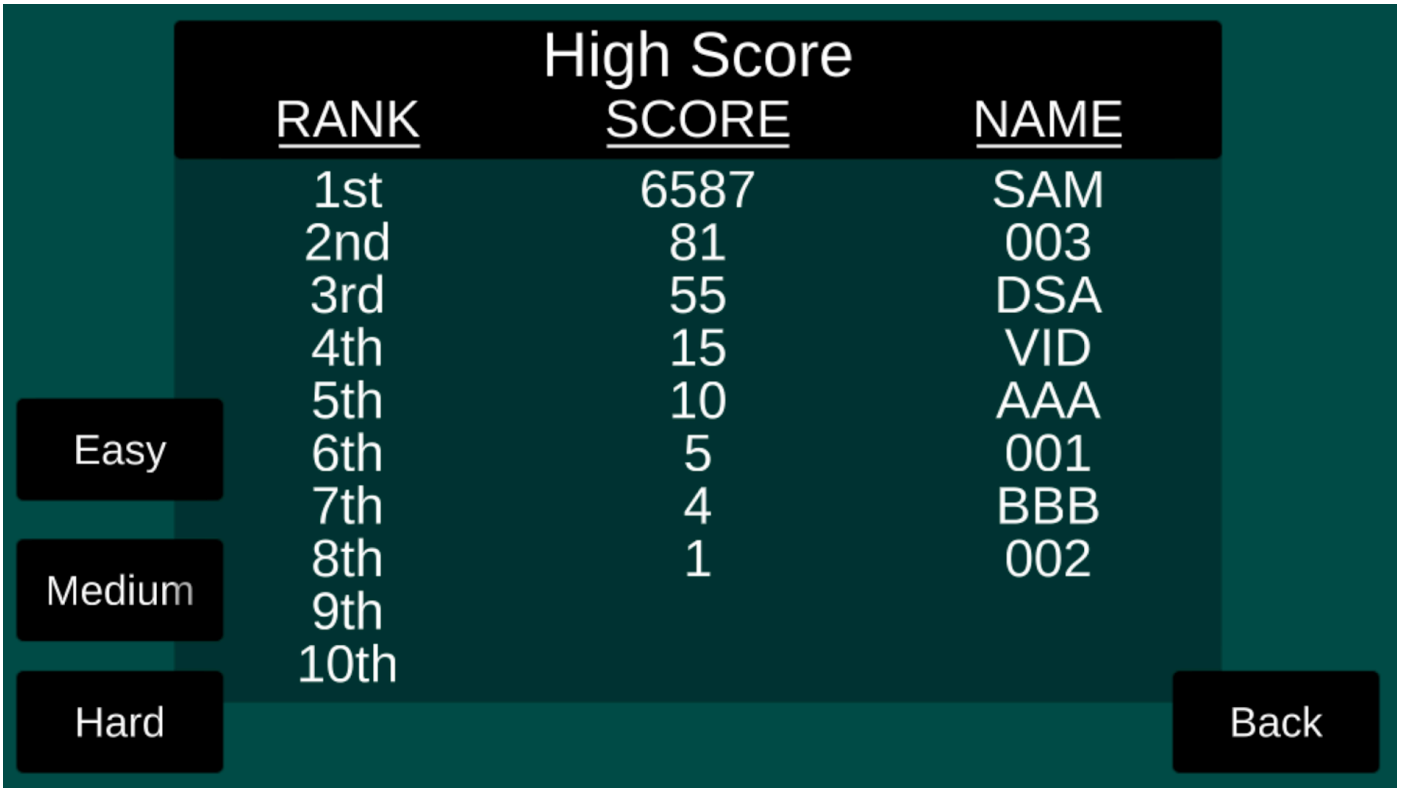
When designing the scoreboard it was initially wanted to show 20+ scores. Although, because of how much room a clearly displayed score takes up on a screen we designed it to only show the top 10.

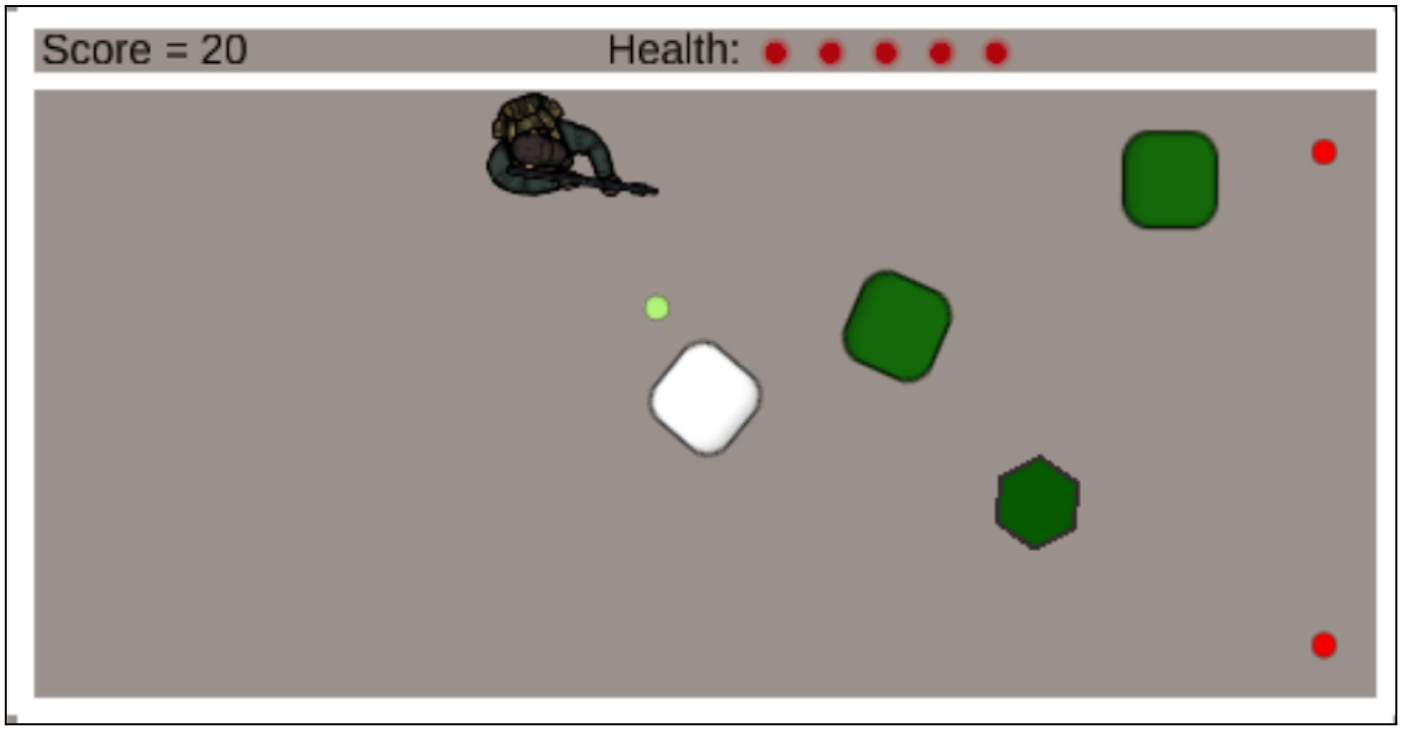
Validation

Validation is the process of checking whether an input is compatible with its purpose. Most inputs in the program are exclusive to a button press. The player instantiates a projectile if and only if the left mouse button is clicked, no further validation needed there. When the player inputs a username to store their score there is length check validation to ensure that it is three letters long (***if (value.Length != 3)***) but as it is stored as a string and i will accept any character no other validation is needed.

Development - The Alpha Build - Formal review

At this stage of the project all designed modules have been developed and tested. All areas of the project seem to work after all white and black box testing and the project is ready to move onto post development testing.





Post Development Testing

Now that all sections have been developed and put through functional testing it was concluded that the game is at a stable build to be tested by external clients - the Alpha Build. An opportunity was carried out by a sample of 10 people, each of these people are non computer scientists, some had a level of gaming intuition but others didn't. This testing found areas of the program that, although work in code with no errors, have issues when it comes to playing and useability.

This is an example of Post Development User Testing:
User Video - https://www.youtube.com/watch?v=ju0Loy9JZQo&feature=youtu.be
Screen recording - https://www.youtube.com/watch?v=HP7OtMr63PM&feature=youtu.be
Most made similar comments and had a very similar experience - below is a summary of their experience.

Walk through of project test - majority comments and problem solution
The players first open on to the menu, all buttons in the menu work correctly and function as intended. As the buttons highlight and change when clicked this was commented by the players as a useful and clear feature. When opening the score board and main game a problem was found.

The first problem found was with the interface and, specifically, the use of a canvas. The Alpha Build was using a form of canvas that didn't scale properly with the size of the screen so when the screen was maximised the canvas didn't scale at the same rate to all the objects and did not scale uniformly around the same mount. This resulted in text appearing in un-intended locations and the scoreboard became a layered mass of objects on top of each other. Due to the properties of my computer, the interface that worked perfectly in 16:9 didn't work for any other scaling. To allow this to work it would mean implementing a completely different UI system and therefore will not be addressed. The game therefore can only run properly for a screen scale of 16:9.

Then the player moved onto the main game. This was noted by the players as smooth and clearly laid out. A negative note was in the lack of tutorial that caught some players in the first game although all 10 eventually worked out the controls on their own. The games collision, instancation and scoring worked as intended although the players found a problem in the design of the game play.

The second problem found addressed in this testing was the game play - specifically the rate of spawning and firing being too fast, resulting in the enemies being too aggressive and the firing having no focus on aim or tactics, just clicking fast, directly contradicting the initial aim, "challenge the players reflexes, strategy and awareness, as well as forcing them to improve over time". First, the spawner was set up to spawn on a round robin style timer system, the first would spawn instantly but the rest would be delayed by X time, the next by 2X time and the 4rd by 3X time. After the spawning the spawner will wait 4X time, such that there is a spawn every X time but at a different place each time. This drastically reduced the rate that enemies spawn allowing for the player to react to each enemy individually whilst having to be aware that more enemies are coming. To address the fire rate a reloading system was implemented.

The reloading system works like the spawning system. 5 rounds in the magazine. Every shot reduces this value by 1. When rounds = 0 or the R key pressed reloading starts, the player cannot shoot while reloading to stop then for permanently reloading. Then a timer ticks down every update and when the timer = 0 the rounds are reset to 5. In addition, there is also a bar that shows visually how long till the weapon is loaded so the player can judge their actions accordingly.

The Bar is a Unity prefab of a scroll bar with the object used to manually score removed, this left a bar that was filled dependent on the value of ***ReloadBar.valueB***.

The C# script “PlayerController” inside the Update() function

```
if (Input.GetKeyDown("r")) //if R key pressed
    {
      tempTime = timeToLoad; //reset timer
      reloading = true; //start reloading
      UpdateAmmo(); //update the ammo in case of change
    }

    if (Input.GetMouseButtonDown(0)) //if left mouse click
    {
      if (rounds > 1 && !reloading) //and there are more than 1 round in the chamber and you are not reloading
      {
        Instantiate(projectile, shootPoint.position, transform.rotation); //instantiate projectile
        rounds--; //reduce rounds by one
        UpdateAmmo(); //and update the ammo display
      }
      else if (rounds == 1 && !reloading) //and there is 1 round in the chamber and you are not alreading reloading
      {
        Instantiate(projectile, shootPoint.position, transform.rotation); //instantiate last projectile
        rounds--; //reduce the rounds by one
        UpdateAmmo(); //update the display
        tempTime = timeToLoad; //reset the timer
        reloading = true; //and start reloading
      }
      else { } //nothing happens if you are reloading
    }

    if (reloading) //if you are reloading
    {
      if (tempTime <= 0) //if reloading timer = 0
      {
        rounds = 5; //refill rounds in magazine
        reloading = false; //stop reloading
        UpdateAmmo(); //update the display
        ReloadBar.value = 0; //and set reloading bar to 0
      }
      else //if time doesn't = 0
      {
        tempTime -= Time.deltaTime * (difficulty * 1); //timer ticks down
        ReloadBar.value = tempTime; //and change the value on the loading bar - visual display to how long reloading takes
      }
    }
```

The Player then completed the game and saved their score, taking them back to the main menu. Some players decided to change the difficulty and play at a different speed while others continued to play on medium. After this most players investigated the scoreboard. Comments on all the features were that they worked clearly and smoothly. No other notable bugs were found.
End of user testing.

Final Testing - against Success Criteria

Compared with the success criteria defined in the analysis - this is done clearly in a video found after this table. The time stamps found in the table link to when in the video the test is made.

Game Play		**Is it Met?**
Player	The player must move in a two dimensional plane with equal movement in all directions	This feature is met. 0:05 - 0:12
	The player must be able to attack with each weapon and damage the enemies	This feature is met to an extent - only one type of weapon is used in game, it works fully. 0:12 - 0:15 and later points
	The player must be killed (the game ends) when they run out of lives	This feature is met. 1:35 - 1:40
Enemies	All enemies must be able to move towards the player in the intended manner	This feature is met. 0:20 - 1:35
	All enemies must be able to attack the player in the intended manner to remove a life on a hit	This feature is met. 0:20 - 1:35
	All enemies must be able to spawn at the intended time in the game (harder enemies are spawned later in the game)	This feature is met. 2:35 - 3:25
	The rate of enemies increase as the game goes on (difficulty must increase)	This feature is met. 2:35 - 3:25
Map	All moving entities should either be blocked or be destroyed when reaching the sides of the map. This is also true of any obstacles within the map	This feature is met. 0:42 - 0:48 and later points
Game Features		
Score	The Score must increase, additively, for every enemy killed (e.g. Score = Score + 1 when chaser killed)	This feature is met. 0:20 - 1:35 and 2:35 - 3:25
	The score must increase multiplicatively for time spent in the game alive, this should be done at the end of the game. (e.g. Score = Score * time when game ends)	This feature is not met
Saving	At end of game, must pick a username for their score to save	The saving isn't mandatory but this feature is met. 1:35, 3:20, 3:50 and 4:00
	The users information along with their score must be saved permanently to the game.	This feature is met. 1:35 - 2:25 and 4:05 onwards
Presenting	Any user should be able to see the top 20 or so scores and usernames, this would be sorted into highest score at the top of the list	This feature is met. Top 10 scores. 4:05 onwards

Final Testing - Solution of the Score problem

The score currently is set to work purely additively, increasing only on player kills although it should be multiplied by time spent alive to be in line with the success criteria. This will be done by having a value to increase every frame of the device and then is multiplied to score at the end.

A timer display is added to the interface in game to show the time alive

```
public void UpdateTime()
{
    DesplayTime.text = "Time: " + timeAlive; //set the text of time display object to “Time: whatever the time is”
}
```

Then every Update the timer increase by the time that has changed since the last update

```
private void Update() //runs every frame
{
    timeAlive = timeAlive + Time.deltaTime; //increases the time alive by the frame rate
    UpdateTime();
```

At the end the score is multiplied by the time alive. This means scores are now dramatically larger but this will add more variety into the scoreboard and will not seem unusual once many games are played.

```
public void EndGame() //runs when player dies
{
    score = score * timeAlive; //score is multiplied by the time alive
    score = Math.Round(score); //time ilive is rounded to nearest whole number
```

Final Testing - Success Criteria Video

The Final Test was carried out to check that the entire program is functional and fulfils the criteria set at the start of the program - the whole test was videoed to show proof of the working solution.
The first part of the video is specifically showing the game tests. It is the main game scene with the spawner objects disabled. This allows for better showing of the player and each enemy individually. Enemy prefabs will be dragged onto the screen and show them being tested. It will also allow the game to be stopped and started quickly to show the prominence of the scoreboard. Then the game will be put back to its full form and the full solution will be shown including the different scoreboards and difficulties.

Use time stamps in the success criteria review to find points in video

https://www.youtube.com/watch?v=E6J0BCG0VRs&feature=youtu.be

Evaluation

The Evaluation will review the completed program and go over what works and what doesn't, what could be added for further development and address any other issues not mentioned in other areas. Nothing in this section will change the program, everything here is already developed or will be suggestions of further development that could happen.
Initially the project will be evaluated against the initial aim.
Then the project will be evaluated against its usability
Then any further development will be addressed, in addition to going over how one of the further developments could be implemented.
The limitations of the programmed solution will then be addressed.
This will be followed by a conclusion on the whole program in a short paragraph.

Evaluation - against the initial aim

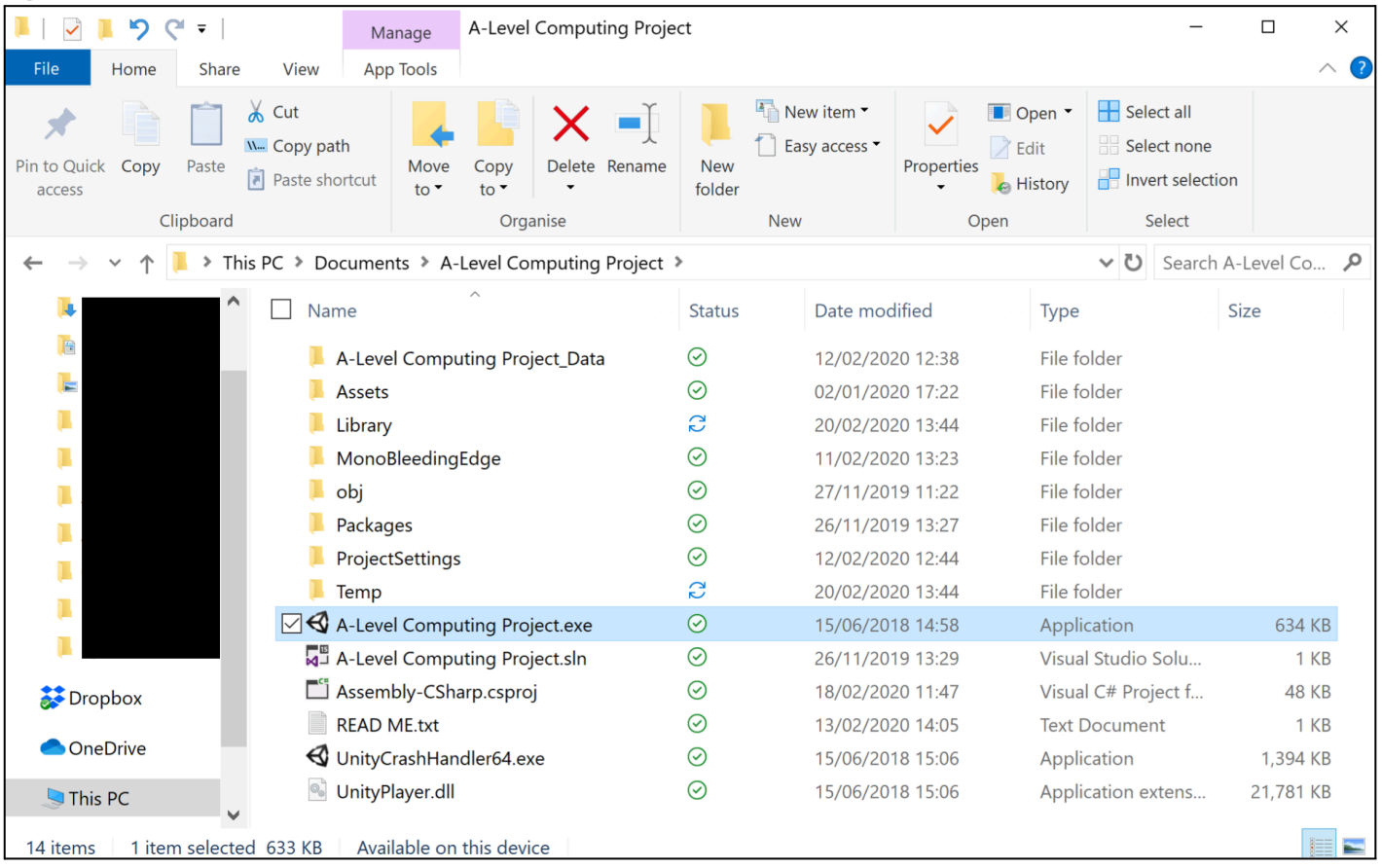
The initial aim - To make a product that “challenges the player’s reflexes, strategy and awareness, as well as forcing them to improve over time” - was the focus of the whole program. The result is a top down shooter that creates an ever changing environment where the player has to adapt to, and must be always aware of what is going on. This can then be replayed any number of times to see improvement, in addition to the option of increasing the game's difficulty to a harder mode allowing for further improvement. Finally, the player performance is recorded to make it as easy as possible for the player to see their improvement.
Noting the success criteria, the point about how the weapon functions has only been partially met due to the lack of multiple weapons. This will be addressed in potential future development.
It is felt that all of these points build to a solution that meets the goal very well.

Evaluation - Usability Features

The game has many usability features.

- The game's menu system works in a very linear fashion and the game has very few inputs. This allows for the game to have very few chances for confusion or invalid inputs. The path though the game should be a clean one with little confusion.
    - When it comes to the game play however there may be confusion. On the one hand the game uses standard inputs that with minimal experimentation can be worked out easily with little to no consequence of failure. Movement is done with both arrow keys and WASD, the weapon is aimed with the mouse so shooting with a click is a clear assumption and if any mistakes are made or realisations coming too late you can just start the game again with no problem.
    - Pressing R reloads the weapon - this is the only point in the game that is hard for the player to find. If future development was made a small tutorial could be implemented to show the player this.
- The color scheme of the game is clear and consistent. Most objects have high contrast with their background allowing for easy identification. In addition, The color scheme works with color blindness - examples of this are featured in figure 2-5 below. For more examples the sight used was https://www.color-blindness.com/coblis-color-blindness-simulator/ and all picures worked with all types of color blindness avalable.
- The Game does not require any external software apart from the download. This allows the game to run on more devices and make installation easier
    - Total size of the game file is roughly 150MB
    - This ‘download’ is not public although if the project were to be published andthe player wanting to play the game, they would have to download all files needed for the game, shown in figure 1

Fig. 1



Examples of different color blind perspectives on the solution

Fig. 2

Fig. 3

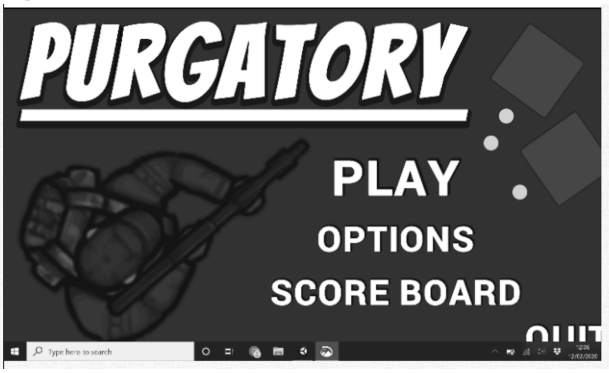


Fig. 4

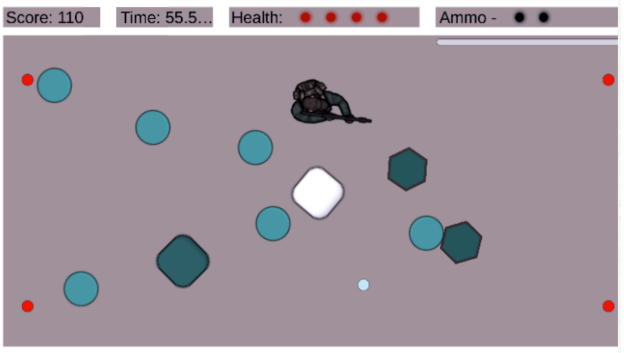
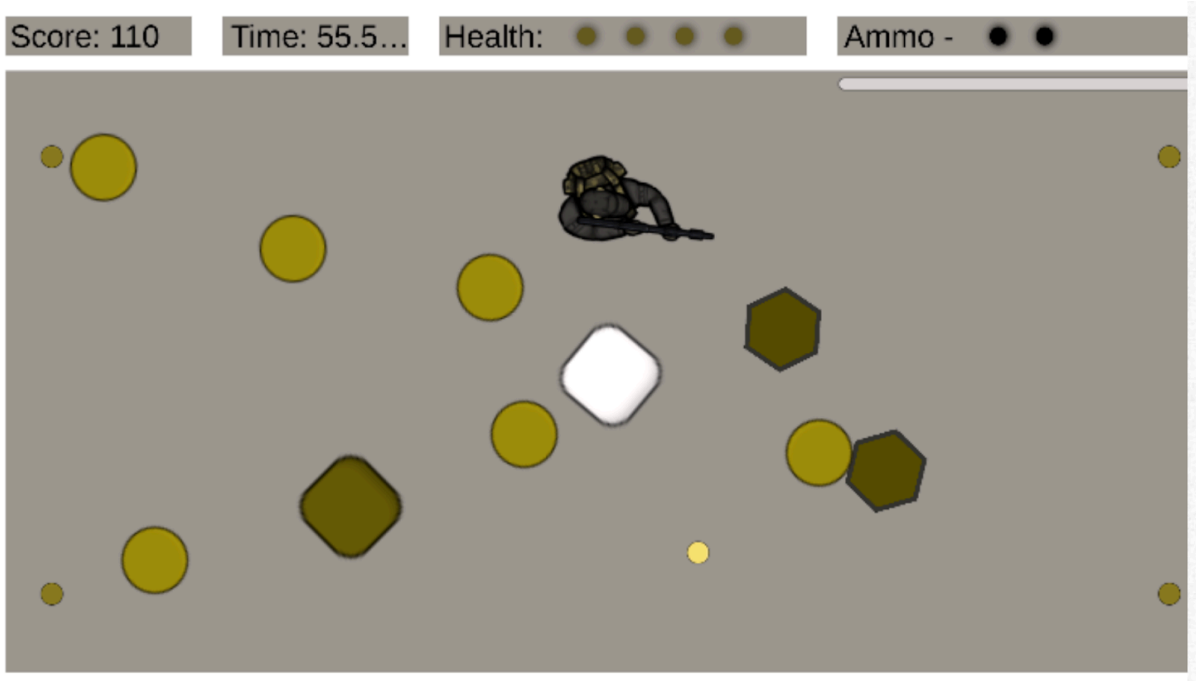


Fig. 5



Evaluation - Further Development

There are many opportunities for further development, many already mentioned.

- Implementation of a tutorial would allow for an easier learning experience when introduced to the game. This feature is even more essential if the game was to increase in complexity and so is highly likely for further development.
- The idea of having many weapons is very possible and would have a large amount of variety in the game. Although given this would be large amounts of repeated code and would likely lead to reworking the score board i will leave it out of the scope of this solution.
    - There is a section bellow explaining how this could be implemented
- Other / more dynamic level design is a potential feature. One thing not explored very deeply and commented on many times by people testing the game was the simplicity of the levels and the use of walls. Walls would be something that move around or can be moved by entities allowing for a highly dynamic gameplay environment appealing to the original idea - this idea could expand to movable or more dynamic spawners to greatly change the state of the environment. Different levels are an option although this would move the game from a score based system to a progress based system and would be difficult to record and compare scores.
- Power ups, collectible items or abilities are all ideas for potential new features. Ways of regaining life, ammunition or extra damage can drastically change games and create very dramatic and memorable moments through your choice of using them. There are many ways these could be implemented from timed spawning like enemies or be dropped by enemies on kills.
- The system in which player scores are saved and loaded uses a very minimal fashion. Only saving the highest 10 scores. To make this system more comprehensive it could be beneficial to save all scores and carry out a sort after every save (most likely bubble) therefore allowing for more scores to be shown. A scroll could be added to the scoreboard to show to the player.
- A further development of the previous point would be saving the data on a server. This would mean that all instances of the game would have the same scoreboard and same level of competition. This system would be highly complex requiring access to an online server location that can run at all times. Because of this, even with further time spent on the project this is unlikely to be feasible for this scale of project.
- Sound could be added to the game. Currently there is no sound or special effects of any kind. Sound and particle effects can bring life to a game although this addition would now add anything to the core design of the game and therefore is not in the scope for the prototype.

Evaluation - Potential solutions to unmet criteria

Currently the weapons works through this code

```
if (Input.GetKeyDown("r")) //if R key pressed
    {
      if (!reloading)
      {
        tempTime = timeToLoad; //reset timer
        reloading = true; //start reloading
        UpdateAmmo(); //update the ammo in case of change
      }
    }

    if (Input.GetMouseButtonDown(0)) //if left mouse click
    {
      if (rounds > 1 && !reloading) //and there are more than 1 round in the magazine and you are not reloading
      {
        Instantiate(projectile, shootPoint.position, transform.rotation); //instantiate projectile
        rounds--; //reduce rounds by one
        UpdateAmmo(); //and update the ammo display
      }
      else if (rounds == 1 && !reloading) //and there is 1 round in the magazine and you are not alreading reloading
      {
        Instantiate(projectile, shootPoint.position, transform.rotation); //instantiate last projectile
        rounds--; //reduce the rounds by one
        UpdateAmmo(); //update the display
        tempTime = timeToLoad; //reset the timer
        reloading = true; //and start reloading
      }
      else { } //nothing happens  if you are reloading
    }

    if (reloading) //if you are reloading
    {
      if (tempTime <= 0) //if reloading timer = 0
      {
        rounds = 5; //refill rounds in magazine
        reloading = false; //stop reloading
        UpdateAmmo(); //update the display
        ReloadBar.value = 0; //and set reloading bar to 0
      }
      else //if time doesn't = 0
      {
        tempTime -= Time.deltaTime * (difficulty); //timer ticks down
        ReloadBar.value = tempTime; //and change the value on the loading bar - visual display to how long reloading
takes
      }
    }
```

If this code were to be surrounded with a decision statement (***If Weapon = 1 {}***) copy if for other paths in the decision (***Else if Weapon = 2 {}***) and then allow the player to choose between the values of weapon in the options menu, that would create different weapons. Then, for each decision in the tree, change some value; For example, a weapon could have more rounds in its magazine and shoot faster but have a much longer reload time. To make this more complex, this value could be passed to the bullet such that it acted differently, allowing it to pass through and damage multiple enemies. Finally, editing could be made to the instantiation code to allow the player to instantiate multiple bullets, to create a buckshot / birdshot type weapon. This weapon could have only one or two rounds in the chamber to balance it.

Evaluation - Maintenance and Software/Hardware limitations

The project has very few maintenance requirements other than download and very few limitations were found in it's hard and software due to its low requirements.

Maintenance - The game requires no setup other than the downloading of the main file and all its contents totaling at 150MB of data. Inside this main file is a READ ME text document that clarifies this and tells the user which item will run the program.

PURGATORY - SB

Thank you for reading this.

This game requires no setup other than downloading the folder that I am contained within and its contents. The .exe named A-Level Computing Project will open a window to configure the game and then from there pressing "Play!" will run the game.

Thank you
Sam

Limitation - due to the program having a small total file size it doesn't result in many limitations. In addition, no other software is needed to run the program.

One limitation of the game is that it can only be played by one player at one time. This although is not a significant limitation nor is it something that would come in further development. The game is designed to challenge a single player and their ability to adapt and improve. Adding in the option for multiple players would move the game away for the initial and therefore will not be implemented.

Evaluation - Conclusion

In conclusion, a full working solution has been developed that satisfies, not just the initial main aim, but has fulfilled the requirements of other more specific requirements. In addition, it includes additional features to make the experience more enjoyable and to more closely align with the initial goal. No significant errors have arisen in the final version of the solution, therefore I deem this project a success.

Appendix

Game

Player Controller

```
using System;
using System.Threading;
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class PlayerController : MonoBehaviour
{
    public float speed = 10; //creates and defines the variable speed. public therefore can change in IDE
    [Space]
    private Rigidbody2D rb; //Creates a variable rb of type 2D Rigid Body
    private Vector2 moveVelocity; //Creates a variable moveVelocity of type 2D Vector
    [Space]
    private Vector3 faceTarget;
    private float offset = -90;
    public GameObject projectile;
    public Transform shootPoint;
    [Space]
    private double Lives = 5;
    private double timeAlive = 0;
    public double score = 0;
    [Space]
    public Image Lives1; //set to the red dots - the life points
    public Image Lives2; //Lives1 set to the first life point, Lives5 set to the last life point
    public Image Lives3;
    public Image Lives4;
    public Image Lives5;
    [Space]
    private int rounds;
    public float timeToLoad;
    private float tempTime;
    private bool reloading;
    [Space]
    public Slider ReloadBar;
    public Image Ammo1; //set to the Black dots - the rounds in magazine
    public Image Ammo2; //Ammo1 set to the first round, Lives5 set to the last round
    public Image Ammo3;
    public Image Ammo4;
    public Image Ammo5;
    [Space]
    public GameObject spawner1;
    public GameObject spawner2;
    public GameObject spawner3;
    public GameObject spawner4;
    [Space]
    public GameObject gameUI;
    public GameObject EndGameUI;
    public GameObject ScoreInputUI;
    [Space]
    public TextMeshProUGUI DesplayScore;
    public TextMeshProUGUI DesplayTime;
    public TextMeshProUGUI DesplayFinalScore;
    [Space]
    private float difficulty = Butten_Controler.difficulty;

    public void EndGame() //runs when player dies
    {
        score = score * timeAlive; //score is multiplied by the time alive
        score = Math.Round(score); //time ilive is rounded to nearest whole number

        spawner1.SetActive(false); //spawners are diabled
        spawner2.SetActive(false);
        spawner3.SetActive(false);
        spawner4.SetActive(false);

        GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enemy"); //any enemies on screen are destroyed
```

```
    foreach (GameObject enemy in enemies)
        GameObject.Destroy(enemy);

    EndGameUI.SetActive(true); //UI changes

    DesplayFinalScore.text = "Your Score: " + score; //the display object desplayer final score
}

public void UpdateScore()
{
    DesplayScore.text = "Score: " + score; //set the text to “Score: whatever the score is”
}

public void UpdateTime()
{
    DesplayTime.text = "Time: " + timeAlive; //set the text of time display object to “Time: whatever the time is”
}

public void UpdateLife() //as  lives decrease
{
    if (Lives <= 4) //objects representing lives are hidden
    {
        Lives1.enabled = false; //hide the life point
    }
    if (Lives <= 3)
    {
        Lives2.enabled = false;
    }
    if (Lives <= 2)
    {
        Lives3.enabled = false;
    }
    if (Lives <= 1)
    {
        Lives4.enabled = false;
    }
    if (Lives <= 0) //when no lives left
    {
        Lives5.enabled = false; //hide the life point
        gameObject.SetActive(false); //player disabled and
        EndGame(); //The game ends
    }
}

public void UpdateAmmo() //as  lives decrease
{
    if (rounds == 5) //objects representing lives are hidden
    {
        Ammo1.enabled = true;
        Ammo2.enabled = true;
        Ammo3.enabled = true;
        Ammo4.enabled = true;
        Ammo5.enabled = true;
    }
    if (rounds == 4)
    {
        Ammo1.enabled = false;
        Ammo2.enabled = true;
        Ammo3.enabled = true;
        Ammo4.enabled = true;
        Ammo5.enabled = true;
    }
    if (rounds == 3)
    {
        Ammo1.enabled = false;
        Ammo2.enabled = false;
        Ammo3.enabled = true;
        Ammo4.enabled = true;
        Ammo5.enabled = true;
    }
    if (rounds == 2)
```

```
    {
        Ammo1.enabled = false;
        Ammo2.enabled = false;
        Ammo3.enabled = false;
        Ammo4.enabled = true;
        Ammo5.enabled = true;
    }
    if (rounds == 1) //when no lives left
    {
        Ammo1.enabled = false;
        Ammo2.enabled = false;
        Ammo3.enabled = false;
        Ammo4.enabled = false;
        Ammo5.enabled = true;
    }
    if (rounds == 0)
    {
        Ammo1.enabled = false;
        Ammo2.enabled = false;
        Ammo3.enabled = false;
        Ammo4.enabled = false;
        Ammo5.enabled = false;
    }
}

private void Awake() //Runs at start of program
{
    gameUI.SetActive(true);
    EndGameUI.SetActive(false);
    ScoreInputUI.SetActive(false);

    rb = GetComponent<Rigidbody2D>(); //define rb to be this components rigid body
    UpdateScore();

    Lives1.enabled = true;
    Lives2.enabled = true;
    Lives3.enabled = true;
    Lives4.enabled = true;
    Lives5.enabled = true;

    Ammo1.enabled = true;
    Ammo2.enabled = true;
    Ammo3.enabled = true;
    Ammo4.enabled = true;
    Ammo5.enabled = true;

    timeAlive = 0;
    rounds = 5;
    reloading = false;
}
```

```
    private void Update() //runs every frame
    {
        timeAlive = timeAlive + Time.deltaTime; //increases the time alive by the frame rate
        UpdateTime();

        Vector2 moveInput = new Vector2(Input.GetAxis("Horizontal"), Input.GetAxis("Vertical")); //define moveInput to be equal to any input of vertical or horizontal direction (arrow keys or WASD) - only happens when the appropriate key pressed
        moveVelocity = moveInput.normalized * speed; //normalizes the vector making the value 1 (just direction) then times it by the speed
        rb.MovePosition(rb.position + moveVelocity * difficulty * Time.fixedDeltaTime); //moves the sprite to its position plus the vector. it's multiplied by Time.fixedDeltaTime so the distance is the same on all processing speeds
        faceTarget = Camera.main.ScreenToWorldPoint(Input.mousePosition); //This Vector3 definition is added at the start - face position is the mouse position
        Vector3 difference = faceTarget - transform.position; //new Vector3 is difference between
        float rotZ = Mathf.Atan2(difference.y, difference.x) * Mathf.Rad2Deg; //new float is how much to rotate

        transform.rotation = Quaternion.Euler(0f, 0f, rotZ + offset); //rotation code, 0 in x and y, rotZ in Z

        if (Input.GetKeyDown("r")) //if R key pressed
        {
            if (!reloading)
            {
                tempTime = timeToLoad; //reset timer
                reloading = true; //start reloading
                UpdateAmmo(); //update the ammo in case of change
            }
        }

        if (Input.GetMouseButtonDown(0)) //if left mouse click
        {
            if (rounds > 1 && !reloading) //and there are more than 1 round in the magazine and you are not reloading
            {
                Instantiate(projectile, shootPoint.position, transform.rotation); //instantiate projectile
                rounds--; //reduce rounds by one
                UpdateAmmo(); //and update the ammo display
            }
            else if (rounds == 1 && !reloading) //and there is 1 round in the magazine and you are not alreading reloading
            {
                Instantiate(projectile, shootPoint.position, transform.rotation); //instantiate last projectile
                rounds--; //reduce the rounds by one
                UpdateAmmo(); //update the display
                tempTime = timeToLoad; //reset the timer
                reloading = true; //and start reloading
            }
            else { } //nothing happens if you are reloading
        }

        if (reloading) //if you are reloading
        {
            if (tempTime <= 0) //if reloading timer = 0
            {
                rounds = 5; //refill rounds in magazine
                reloading = false; //stop reloading
                UpdateAmmo(); //update the display
                ReloadBar.value = 0; //and set reloading bar to 0
            }
            else //if time doesn't = 0
            {
                tempTime -= Time.deltaTime * (difficulty); //timer ticks down
                ReloadBar.value = tempTime; //and change the value on the loading bar - visual display to how long reloading takes
            }
        }

        if (Input.GetKeyDown("p"))
        {
            Thread.Sleep(5000);
        }
    }
```

```
    private void OnTriggerEnter2D(Collider2D collision) //if collision reduce the health by 1
    {
        if (collision.CompareTag("Enemy")) //multiple tags doing same thing here will do different things later
        {
            Lives = Lives - 1f; //reduces lives by one
            Debug.Log("Lives = " + Lives);
            UpdateLife(); //Life is updated after a collision as that is when life changes
        }
        if (collision.CompareTag("Projectile"))
        {
            Lives = Lives - 0.5f; //reduces lives by half due to double collisions
            Debug.Log("Lives = " + Lives);
            UpdateLife();
        }
        if (collision.CompareTag("EnemyProjectile"))
        {
            Lives = Lives - 0.5f; //reduces lives by half due to double collisions
            Debug.Log("Lives = " + Lives);
            UpdateLife();
        }
    }
}
```

EnemyChaser

```
using UnityEngine;


public class EnemyChaser : MonoBehaviour
{
    public float speed = 5; //variable making speed of the enemy
    public int Value = 1; //variable dictating the value of killing this
    [Space]
    private Transform target; //set a variable to be later defined as the location of the player
    [Space]
    private Vector3 faceTarget;
    private float offset = -90;
    [Space]
    GameObject player; //code to link with the PlayerController code so I can edit variables
    PlayerController PlayerController;
    [Space]
    private float difficulty = Butten_Controler.difficulty;

    void Awake()
    {
        target = GameObject.FindGameObjectWithTag("Player").GetComponent<Transform>(); //defines the game point target as a sprite with the tag “player”’s location (the player)

        player = GameObject.FindGameObjectWithTag("Player"); //code to link with the PlayerController
        PlayerController = player.GetComponent<PlayerController>();

    }

    void Update()
    {
        transform.position = Vector2.MoveTowards(transform.position, target.position, speed * Time.deltaTime * (difficulty * 1)); //every update the enemy moves towards the target location (the player)

        faceTarget = GameObject.FindGameObjectWithTag("Player").transform.position; //Vector3 position of player
        Vector3 difference = faceTarget - transform.position; //difference between player's location and shooters location
        float rotZ = Mathf.Atan2(difference.y, difference.x) * Mathf.Rad2Deg; //calculate angular difference difference player and shooter

        transform.rotation = Quaternion.Euler(0f, 0f, rotZ + offset); //turn by the angle to face player
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            Destroy(gameObject); //destroy this object when it hits the player
        }
```

```
        if (collision.CompareTag("Projectile"))
        {
            Destroy(gameObject); //destroy this object when hit by projectile
            PlayerController.score = PlayerController.score + Value; //and increase the score by its value
            Debug.Log("Score = " + PlayerController.score);
            PlayerController.UpdateScore(); //update the score after a collision between enemy and player projectile
        }
        if (collision.CompareTag("EnemyProjectile"))
        {
            Destroy(gameObject); //destroy without increasing score if hit with enemy projectile
        }
    }
}
```

EnemyHunter

```
using UnityEngine;


public class EnemyHunter : MonoBehaviour
{
    public float speed = 5; //variable making speed of the enemy
    public int Value = 10; //variable dictating the value of killing this
    [Space]
    private Transform target; //set a variable to be later defined as the location of the player
    [Space]
    private Vector3 faceTarget;
    private float offset = -90;
    [Space]
    GameObject player; //code to link with the PlayerController code so we can edit variables
    PlayerController PlayerController;
    [Space]
    private float difficulty = Butten_Controler.difficulty;

    void Awake()
    {
        target = GameObject.FindGameObjectWithTag("Player").GetComponent<Transform>(); //defines the game point target as a sprite with the tag “player”’s location (the player)

        player = GameObject.FindGameObjectWithTag("Player"); //code to link with the PlayerController
        PlayerController = player.GetComponent<PlayerController>();
    }

    void Update()
    {
        transform.position = Vector2.MoveTowards(transform.position, target.position, speed * Time.deltaTime * (difficulty * 1)); //every update the enemy moves towards the target location (the player)

        faceTarget = GameObject.FindGameObjectWithTag("Player").transform.position; //Vector3 position of player
        Vector3 difference = faceTarget - transform.position; //difference between player's location and shooters location
        float rotZ = Mathf.Atan2(difference.y, difference.x) * Mathf.Rad2Deg; //calculate angular difference difference player and shooter

        transform.rotation = Quaternion.Euler(0f, 0f, rotZ + offset); //turn by the angle to face player
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            Destroy(gameObject); //destroy this object when it hits the player
        }
        if (collision.CompareTag("Projectile"))
        {
            Destroy(gameObject); //destroy this object when hit by projectile
            PlayerController.score = PlayerController.score + Value; //and increase the score by its value
            Debug.Log("Score = " + PlayerController.score);
            PlayerController.UpdateScore(); //update the score after a collision between enemy and player projectile
        }

        if (collision.CompareTag("EnemyProjectile"))
        {
```

```
            Destroy(gameObject); //destroy without increasing score if hit with enemy projectile
        }
    }
}
```

Enemy Shooter

```
using UnityEngine;


public class EnemyShooter : MonoBehaviour
{
    public float speed = 5; //variable making speed of the enemy
    public int Value = 5; //variable dictating the value of killing this
    public float stoppingDistance = 6.5f; //the f signifies it is a float not an integer
    public float retreatDistance = 5; //the difference between the stopping and retreating distance is the place where the shooter can sit, not to close not too far away.The shooter doesn't have to be here to shoot or land a hit, it can shoot anywhere and projectiles should travel indefinitely before collision with wall
    [Space]
    private float timeBtwShots;
    private float startTimeBtwShots = 3;
    [Space]
    public GameObject Projectile;
    public Transform shotPoint;
    private Transform moveTarget;
    private Vector3 shootTarget;
    [Space]
    GameObject player; //code to link with the PlayerController code so I can edit variables
    PlayerController PlayerController;
    [Space]
    private float difficulty = Butten_Controler.difficulty;

    private float offset = -90;

    void Awake()
    {
        moveTarget = GameObject.FindGameObjectWithTag("Player").GetComponent<Transform>(); //Players location

        player = GameObject.FindGameObjectWithTag("Player"); //code to link with the PlayerController
        PlayerController = player.GetComponent<PlayerController>();

        timeBtwShots = startTimeBtwShots;
    }

    void Update()
    {
        if (Vector2.Distance(transform.position, moveTarget.position) > stoppingDistance) //if too far away
        {
            transform.position = Vector2.MoveTowards(transform.position, moveTarget.position, speed * Time.deltaTime * (difficulty * 1)); //move closer
        }
        else if (Vector2.Distance(transform.position, moveTarget.position) < stoppingDistance && Vector2.Distance(transform.position, moveTarget.position) > retreatDistance) //if close enough and far away enough
        {
            transform.position = this.transform.position; //stay still
        }
        else if (Vector2.Distance(transform.position, moveTarget.position) < retreatDistance) //if too close
        {
            transform.position = Vector2.MoveTowards(transform.position, moveTarget.position, -speed * Time.deltaTime * (difficulty * 1)); //move away - note -speed
        }

        if (timeBtwShots <= 0) //when timer = 0
        {
            Instantiate(Projectile, shotPoint.position, transform.rotation); //shoot
            timeBtwShots = startTimeBtwShots; //then reset timer
        }


        else
        {
            timeBtwShots -= Time.deltaTime; //timer ticks down
```

```
        }

        shootTarget = GameObject.FindGameObjectWithTag("Player").transform.position; //Vector3 position of player
        Vector3 difference = shootTarget - transform.position; //difference between player's location and shooters location
        float rotZ = Mathf.Atan2(difference.y, difference.x) * Mathf.Rad2Deg; //calculate angular difference difference player and shooter

        transform.rotation = Quaternion.Euler(0f, 0f, rotZ + offset); //turn by the angle to face player
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            Destroy(gameObject); //destroy this object when it hits the player
        }
        if (collision.CompareTag("Projectile"))
        {
            Destroy(gameObject); //destroy this object when hit by projectile
            PlayerController.score = PlayerController.score + Value; //and increase the score by its value
            Debug.Log("Score = " + PlayerController.score);
            PlayerController.UpdateScore(); //update the score after a collision between enemy and player projectile
        }
        if (collision.CompareTag("EnemyProjectile"))
        {
            Destroy(gameObject); //destroy without increasing score if hit with enemy projectile
        }
    }
}
```

Projectile

```
using UnityEngine;


public class Projectile : MonoBehaviour
{
    public float speed = 50; //projectile speed - should be the fastest thing in game
    [Space]
    private float difficulty = Butten_Controler.difficulty;

    void Update()
    {
        transform.Translate(Vector2.up * speed * difficulty * Time.deltaTime); //move forward
    }

    private void OnTriggerEnter2D(Collider2D collision) //if collide with anything
    {
        if (collision.CompareTag("Player"))
        {
            Destroy(gameObject); //destroy self
            Debug.Log("Hit Player"); //and notify - score calculations happen in enemies
        }
        else if (collision.CompareTag("Enemy"))
        {
            Destroy(gameObject);
            Debug.Log("Hit Enemy");
        }
        else if (collision.CompareTag("Wall"))
        {
            Destroy(gameObject);
            Debug.Log("Hit Wall");
        }
    }
}
```

Spawner

```
using UnityEngine;


public class Spawner : MonoBehaviour
{
    private float difficulty = Butten_Controler.difficulty;
    [Space]
    public GameObject enemy1; //Game objects will be given the prefabricated objects of the enemies.
    public GameObject enemy2; //enemy1 = chaser, enemy2 = shooter, enemy3 = hunter
    public GameObject enemy3;
    [Space]
    private int count = 0; //count the number of time enemies have spawned to allow harder enemies to spawn later in game when count is high
    private float ShooterStart = 30; //number of spawns shooters will start
    private float HunterStart = 50; //number of spawns hunters will start
    [Space]
    private int randNum; //placeholder for a randomly generated number
    [Space]
    public float timeBtwSpawn = 0; //time until next spawn - counts down through the program
    public float startTimeBtwSpawn = 5; //the number timeBtwSpawns is set to after a spawn
    public float timeChange = 0.05f; //the amount startTimeBtwSpawn changes every spawn (decrease)

    private void Awake()
    {
        ShooterStart = 14 - (difficulty * 10);
        HunterStart = 24 - (difficulty * 10);
    }

    private void Spawn(int phase) //function that will spawn a random enemy
    {
        phase++; //the “phase” determines what enemies can spawn - phase++ is faze +1 to make the maths work
        randNum = Random.Range(1, phase); //spawns just enemy1

        //randNum = 1;
        //randNum = 2;
        //randNum = 3;

        if (randNum == 1)
        {
            Instantiate(enemy1, transform.position, Quaternion.identity);
        }
        else if (randNum == 2) //spawns enemy1 or enemy2
        {
            Instantiate(enemy2, transform.position, Quaternion.identity);
        }
        else if (randNum == 3) //spawns any enemy
        {
            Instantiate(enemy3, transform.position, Quaternion.identity);
        }
    }

    void Update()
    {
        if (timeBtwSpawn <= 0) //if the count down to spawn = 0
        {
            count = count + 1; //count the spawn

            timeBtwSpawn = startTimeBtwSpawn; //reset the clock
            if (startTimeBtwSpawn > 1.5f)
            {
                startTimeBtwSpawn = startTimeBtwSpawn - timeChange; //decrease the max on the clock
            }

            if (count <= ShooterStart) //start of the game
            {
                Spawn(1); //phase 1
            }


            else if (count > ShooterStart && count <= HunterStart)
```

```
            {
                Spawn(2); //phase 2
            }
            else if (count > HunterStart) //late game
            {
                Spawn(3); //phase 3
            }
        }
        else //if time doesn't = 0
        {
            timeBtwSpawn -= Time.deltaTime * (difficulty * 1); //timer ticks down
        }
    }
}
```

Button Controller

```
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using TMPro;

public class Butten_Controler : MonoBehaviour {

    static public float difficulty = 1f;
    public bool OnHighscoreTable; //the code used to save data to the list is used in multiple scenes, therefore their score table can only be updated if you are on the high score table, therefore a public bool is used to show this.
    [Space]
    public GameObject game; //a series of variables that represent various object
    public GameObject gameUI;
    public GameObject EndGameUI;
    public GameObject ScoreInputUI;
    [Space]
    public TextMeshProUGUI ScoreDesctiption;
    public TextMeshProUGUI ScoreNotValid;
    [Space]
    public Button SaveScore_btn;
    public TMP_InputField InputName;
    [Space]
    public PlayerController PlayerController;
    public Scoreboard Scoreboard;
    [Space]
    private ScoreboardEntryData NewScore;
    [Space]
    public Canvas Scoreboard_Easy;
    public Canvas Scoreboard_Medium;
    public Canvas Scoreboard_Hard;

    private void Awake()
    {
        if (OnHighscoreTable) //if on the high score screen
        {
            Scoreboard_Easy.enabled = false;
            Scoreboard_Medium.enabled = true;
            Scoreboard_Hard.enabled = false;
        }
    }

    public void MainMenu() //when button pressed this function is called
    {
        SceneManager.LoadScene(0); //Scene is changed - MainMenu
        Debug.Log("Back Button Pressed");
    }

    public void Options()
    {
        SceneManager.LoadScene(1); //Scene is changed - OptionsMenu
        Debug.Log("Options Button Pressed");
    }

    public void Score()
    {
```

```
    SceneManager.LoadScene(2);  //Scene is changed - Scoreboard
    Debug.Log("Score Button Pressed");
}

public void Play()
{
    SceneManager.LoadScene(3);  //Scene is changed - Main Game
    Debug.Log("Play Button Pressed");
}

public void Easy()
{
    difficulty = 0.5f; //the value is changed
    Debug.Log("Easy Button Pressed");
    Debug.Log(difficulty);
}

public void Medium()
{
    difficulty = 1f; //the value is changed
    Debug.Log("Medium Button Pressed");
    Debug.Log(difficulty);
}

public void Hard()
{
    difficulty = 2f; //the value is changed
    Debug.Log("Hard Button Pressed");
    Debug.Log(difficulty);
}

public void Scores_Easy()
{
    Scoreboard_Easy.enabled = true;
    Scoreboard_Medium.enabled = false;
    Scoreboard_Hard.enabled = false;
}

public void Scores_Medium()
{
    Scoreboard_Easy.enabled = false;
    Scoreboard_Medium.enabled = true;
    Scoreboard_Hard.enabled = false;
}

public void Scores_Hard()
{
    Scoreboard_Easy.enabled = false;
    Scoreboard_Medium.enabled = false;
    Scoreboard_Hard.enabled = true;
}

public void Quit()
{
    Application.Quit(); //End the program
    Debug.Log("Quit Button Pressed");
}
```

```
    public void SaveScoreScreen()
    {
        game.SetActive(false); //activate and deactivate objects to create the screen
        gameUI.SetActive(false);
        EndGameUI.SetActive(false);

        ScoreInputUI.SetActive(true);

        ScoreDesctiption.enabled = true;
        ScoreNotValid.enabled = false;
    }

    public void SaveScore()
    {
        ScoreDesctiption.enabled = false;

        string value = InputName.text;

        if (value.Length != 3)
        {
            ScoreNotValid.enabled = true;
        }
        else
        {
            SaveScore_btn.enabled = false;

            NewScore.entryScore = PlayerController.score;
            NewScore.entryName = InputName.text.ToUpper();

            if(difficulty == 0.5f)
            {
                Scoreboard.AddEntry_E(NewScore);
            }
            else if(difficulty == 1f)
            {
                Scoreboard.AddEntry_M(NewScore);
            }
            else if(difficulty == 2f)
            {
                Scoreboard.AddEntry_H(NewScore);
            }

            SceneManager.LoadScene(0);
        }
    }
}
```

Enable_Game

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Enable_Game : MonoBehaviour {

    public GameObject game;

    void Awake ()
    {
        game.SetActive(true);
    }
}
```

ScoreBoard

```
using System.IO;
using UnityEngine;

public class Scoreboard : MonoBehaviour
{
    [SerializeField] private int maxScoreboardEntries = 10; //max number of scores in list
    public bool OnHighscoreTable; //the code used to save data to the list is used in multiple scenes, therefore their score table can only be updated if you are on the high score table, therefore a public bool is used to show this.
    [Space]
    [Header("EASY")]
    [SerializeField] private Transform highscoresHolderTranform_E; //location of where the scores should be instantiated
    [SerializeField] private GameObject scoreboardEntryObject_E; //object to instantiate objects into

    [Space]
    [Header("MEDIUM")]
    [SerializeField] private Transform highscoresHolderTranform_M; //location of where the scores should be instantiated
    [SerializeField] private GameObject scoreboardEntryObject_M; //object to instantiate objects into

    [Space]
    [Header("HARD")]
    [SerializeField] private Transform highscoresHolderTranform_H; //location of where the scores should be instantiated
    [SerializeField] private GameObject scoreboardEntryObject_H; //object to instantiate objects into

    private string SavePath_E => $"{Application.persistentDataPath}/Highscores_E.json"; //Save location of the text file
    private string SavePath_M => $"{Application.persistentDataPath}/Highscores_M.json"; //Save location of the text file
    private string SavePath_H => $"{Application.persistentDataPath}/Highscores_H.json"; //Save location of the text file

    private void Awake() //runs when the scoreboard scene is opened
    {
        if (OnHighscoreTable)
        {
            ScoreboardSaveData savedScores_E = GetSavedScores_E();
            ScoreboardSaveData savedScores_M = GetSavedScores_M();
            ScoreboardSaveData savedScores_H = GetSavedScores_H();

            UpdateUI_E(savedScores_E);
            UpdateUI_M(savedScores_M);
            UpdateUI_H(savedScores_H);
        }
    }

    public void AddEntry_E(ScoreboardEntryData scoreboardEntryData)
    {
        ScoreboardSaveData savedScores = GetSavedScores_E();

        bool scoreAdded = false; //does the item fit in the list

        for (int i = 0; i < savedScores.highscores.Count; i++) //repeat for the length of the list
        {
            if(scoreAdded)
            {
                Break; //stop comparing values
            }

            if (scoreboardEntryData.entryScore > savedScores.highscores[i].entryScore) //if bigger any value - starting from the first one
            {
                savedScores.highscores.Insert(i, scoreboardEntryData); //insert it in the list at the right place
                scoreAdded = true; //not that the value is in the list
            }
        }

        if (!scoreAdded && savedScores.highscores.Count < maxScoreboardEntries) //if there is less then the max values and the value is not already added, add value to the end of the list
        {
            savedScores.highscores.Add(scoreboardEntryData);
        }


        if (savedScores.highscores.Count > maxScoreboardEntries) //if there are too many items in the list, remove the extra items
```

```
        {
            savedScores.highscores.RemoveRange(maxScoreboardEntries, savedScores.highscores.Count -
maxScoreboardEntries);
        }

        SaveScores_E(savedScores); //save the new list to the text file

        if (OnHighscoreTable) //if on the high score screen
        {
            UpdateUI_E(savedScores); //load and update the table
        }
    }

    public void AddEntry_M(ScoreboardEntryData scoreboardEntryData)
    {
        ScoreboardSaveData savedScores = GetSavedScores_M();

        bool scoreAdded = false; //does the item fit in the list

        for (int i = 0; i < savedScores.highscores.Count; i++) //repeat for the length of the list
        {
            if (scoreboardEntryData.entryScore > savedScores.highscores[i].entryScore) //if bigger any value - starting from the
first one
            {
                savedScores.highscores.Insert(i, scoreboardEntryData); //insert it in the list at the right place
                scoreAdded = true; //not that the value is in the list
                break; //stop comparing values
            }
        }

        if (!scoreAdded && savedScores.highscores.Count < maxScoreboardEntries) //if there is less then the max values and
the value is not already added, add value to the end of the list
        {
            savedScores.highscores.Add(scoreboardEntryData);
        }

        if (savedScores.highscores.Count > maxScoreboardEntries) //if there are too many items in the list, remove the extra items
        {
            savedScores.highscores.RemoveRange(maxScoreboardEntries, savedScores.highscores.Count -
maxScoreboardEntries);
        }

        SaveScores_M(savedScores); //save the new list to the text file

        if (OnHighscoreTable) //if on the high score screen
        {
            UpdateUI_M(savedScores); //load and update the table
        }
    }

    public void AddEntry_H(ScoreboardEntryData scoreboardEntryData)
    {
        ScoreboardSaveData savedScores = GetSavedScores_H();

        bool scoreAdded = false; //does the item fit in the list

        for (int i = 0; i < savedScores.highscores.Count; i++) //repeat for the length of the list
        {
            if (scoreboardEntryData.entryScore > savedScores.highscores[i].entryScore) //if bigger any value - starting from the
first one
            {
                savedScores.highscores.Insert(i, scoreboardEntryData); //insert it in the list at the right place
                scoreAdded = true; //not that the value is in the list
                break; //stop comparing values
            }
        }

        if (!scoreAdded && savedScores.highscores.Count < maxScoreboardEntries) //if there is less then the max values and
the value is not already added, add value to the end of the list
        {
            savedScores.highscores.Add(scoreboardEntryData);
```

```
        }

        if (savedScores.highscores.Count > maxScoreboardEntries) //if there are too many items in the list, remove the extra items
        {
            savedScores.highscores.RemoveRange(maxScoreboardEntries, savedScores.highscores.Count -
maxScoreboardEntries);
        }

        SaveScores_H(savedScores); //save the new list to the text file

        if (OnHighscoreTable) //if on the high score screen
        {
            UpdateUI_H(savedScores); //load and update the table
        }
    }

    private void UpdateUI_E(ScoreboardSaveData savedScores)
    {
        foreach (Transform child in highscoresHolderTranform_E)
        {
            Destroy(child.gameObject); //remove the existing values
        }

        foreach (ScoreboardEntryData highscore in savedScores.highscores)
        {
            Instantiate(scoreboardEntryObject_E,
highscoresHolderTranform_E).GetComponent<ScoreboardEntryUI>().Initialise(highscore); //instantiate objects to values to
the list
        }
    }

    private void UpdateUI_M(ScoreboardSaveData savedScores)
    {
        foreach (Transform child in highscoresHolderTranform_M)
        {
            Destroy(child.gameObject); //remove the existing values
        }

        foreach (ScoreboardEntryData highscore in savedScores.highscores)
        {
            Instantiate(scoreboardEntryObject_M,
highscoresHolderTranform_M).GetComponent<ScoreboardEntryUI>().Initialise(highscore); //instantiate objects to values to
the list
        }
    }

    private void UpdateUI_H(ScoreboardSaveData savedScores)
    {
        foreach (Transform child in highscoresHolderTranform_H)
        {
            Destroy(child.gameObject); //remove the existing values
        }

        foreach (ScoreboardEntryData highscore in savedScores.highscores)
        {
            Instantiate(scoreboardEntryObject_H,
highscoresHolderTranform_H).GetComponent<ScoreboardEntryUI>().Initialise(highscore); //instantiate objects to values to
the list
        }
    }

    private ScoreboardSaveData GetSavedScores_E() //to save data
    {
        if (!File.Exists(SavePath_E)) //if there is no existing table
        {
            File.Create(SavePath_E).Dispose(); //create a file
            return new ScoreboardSaveData(); //and add return the empty list
        }
        using (StreamReader stream = new StreamReader(SavePath_E)) //reading from the list
        {
            string json = stream.ReadToEnd(); //read the values into a string ‘json’
```

```
            return JsonUtility.FromJson<ScoreboardSaveData>(json); //return values from the list
        }
    }

    private ScoreboardSaveData GetSavedScores_M() //to save data
    {
        if (!File.Exists(SavePath_M)) //if there is no existing table
        {
            File.Create(SavePath_M).Dispose(); //create a file
            return new ScoreboardSaveData(); //and add return the empty list
        }

        using (StreamReader stream = new StreamReader(SavePath_M)) //reading from the list
        {
            string json = stream.ReadToEnd(); //read the values into a string 'json'

            return JsonUtility.FromJson<ScoreboardSaveData>(json); //return values from the list
        }
    }

    private ScoreboardSaveData GetSavedScores_H() //to save data
    {
        if (!File.Exists(SavePath_H)) //if there is no existing table
        {
            File.Create(SavePath_H).Dispose(); //create a file
            return new ScoreboardSaveData(); //and add return the empty list
        }

        using (StreamReader stream = new StreamReader(SavePath_H)) //reading from the list
        {
            string json = stream.ReadToEnd(); //read the values into a string 'json'

            return JsonUtility.FromJson<ScoreboardSaveData>(json); //return values from the list
        }
    }

    private void SaveScores_E(ScoreboardSaveData scoreboardSaveData)
    {
        using (StreamWriter stream = new StreamWriter(SavePath_E)) //Writing a sting
        {
            string json = JsonUtility.ToJson(scoreboardSaveData, true); //link the values to write into a string json
            stream.Write(json); //write values into string
        }
    }

    private void SaveScores_M(ScoreboardSaveData scoreboardSaveData)
    {
        using (StreamWriter stream = new StreamWriter(SavePath_M)) //Writing a sting
        {
            string json = JsonUtility.ToJson(scoreboardSaveData, true); //link the values to write into a string json
            stream.Write(json); //write values into string
        }
    }

    private void SaveScores_H(ScoreboardSaveData scoreboardSaveData)
    {
        using (StreamWriter stream = new StreamWriter(SavePath_H)) //Writing a sting
        {
            string json = JsonUtility.ToJson(scoreboardSaveData, true); //link the values to write into a string json
            stream.Write(json); //write values into string
        }
    }
}
```

ScoreboardEntryData

```
using System;
```

```
[Serializable]

public struct ScoreboardEntryData
{
    public string entryName;
    public double entryScore;
}
```

ScoreboardEntryUI

```
using UnityEngine;
using TMPro;


public class ScoreboardEntryUI : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI entryNameText = null;
    [SerializeField] private TextMeshProUGUI entryScoreText = null;

    public void Initialise(ScoreboardEntryData ScoreboardEntryData)
    {
        entryNameText.text = ScoreboardEntryData.entryName;
        entryScoreText.text = ScoreboardEntryData.entryScore.ToString();
    }
}
```

ScoreboardSaveData

```
using System;
using System.Collections.Generic;


[Serializable]

public class ScoreboardSaveData
{
    public List<ScoreboardEntryData> highscores = new List<ScoreboardEntryData>();
}
```

Bibliography

YouTube Videos

- Blackthornprod, 10 Jan 2018. *HOW TO MAKE A 2D CHARACTER CONTROLLER IN UNITY - EASY TUTORIAL*. Youtube - https://www.youtube.com/watch?v=CeXAiaQOzmY&t=0s
- Blackthornprod, 20 Dec 2017. *2D FOLLOW AI WITH UNITY AND C# - EASY TUTORIAL.* Youtube - https://www.youtube.com/watch?v=rhoQd6IAtDo&t=0s
- Blackthornprod, 28 Dec 2017. *SHOOTING/FOLLOW/RETREAT ENEMY AI WITH UNITY AND C#.* Youtube - https://www.youtube.com/watch?v=_Z1t7MNk0c4&t=0s
- Blackthornprod, 10 Aug 2018. *HOW TO MAKE A 2D RANGED COMBAT SYSTEM - UNITY TUTORIAL.* Youtube - https://www.youtube.com/watch?v=bY4Hr2x05p8&t=0s
- Dapper Dino, 19 Jun 2019. *How To Create A Scoreboard System For Your Game - Unity Tutorial.* Youtube - https://www.youtube.com/watch?v=FSEbPxf0kfs&t=0s

Sprites used

- OpenGameArt.org, 19 Jun 2019. opengameart.org - https://opengameart.org/content/animated-top-down-survivor-player