A2 Computer Science Programming project

Snake Revamped

By Rohan Tantepudi (Exam Number 6115, Exam Centre 14285)

Contents

Section 1: Analysis

Identifying the problem

The Problem

Currently, many people in the world struggle to find time to play video games. For example, students (especially in Years 11, 12 and 13) may find that their time is being spent doing work and revising, so much so that time simply cannot be spent playing games with long-winded stories and complex plotlines. The fact is, for some, sitting down and actually committing to playing video games can seem like a long and arduous process, while for others, playing video games is simply part of their daily routine. However, it can be proven that the action of playing video games can be correlated to feelings of relaxation and reduced hostility. In the "Hitman Study" conducted by Christopher J. Ferguson, an American Psychologist at Stetson University in Florida, 103 young adults were chosen (of which 10% women and 8% men never played a violent video game before) to participate in a PASAT task (specifically designed to be frustrating to those who undergo it), before being asked to play either no game, a non-violent game (like Madden 20), a video game with the player playing as the good guy (like Call of Duty 2), or a game played as a bad guy (like Hitman). The test concluded that "Results do not support a link between violent video games and aggressive behaviour, but do suggest that violent games reduce depression and hostile feelings in players through mood management.". This study proves that video games can in fact help people who may be undergoing some form of stress as a way of relaxing and relieving this stress.

However, over the coming years, as technology has improved and computers have become more powerful, game companies are now being able to develop far more complex games than what they could make in the 90s, for example. Now that computers could have more memory and general processing power, environments could become more intricate, characters could be more fleshed out, and stories in games could be longer and could feature more complexities within them. This can be seen in open world titles such as Nintendo's *The legend of Zelda: Breath of the Wild* gaining a very large popularity all around the world. One downside to this, however, is that with increased complexity, comes increased time commitment. More and more single player games nowadays take a much longer time to complete, and the sheer amount of time the average user has to put in to even make it past the base story is massive. For example, the aforementioned *Breath of the Wild* has an average completion time of 50 hours – and that's just the base story; those looking to fully 100% complete the game are looking at spending upwards of 190 hours of gameplay. Even in the multiplayer side of gaming, some popular titles such as Valve's *Counter Strike: Global Offensive* have average match times of over an hour. Other titles rely on longer play times to thrive in the industry, pushing out constant updates to try and keep active players satisfied, and even pushing monetisation methods that nudge people who commit to the game to eventually spend money in it, forcing users to believe that if they don't spend a large amount of time on the game, the money spent will seem wasted (this is typically seen in the mobile gaming industry).

Even though more hours of gameplay can seem like a good thing (and it is), allowing for players to feel like the money spent on the game is money well spent, as well as feeling confident in the quality of the products made by the game developing companies, for some people, spending around £60 on a game you will only play for, say, 30 minutes at a time can seem a waste of time and money. Therefore, I am planning on making a game that is both fun and easy to play (allowing for anyone to play it without needing too much knowledge on how to play well), as well as one that doesn't require a large time commitment. This can mean that people (like students, for example) can use

this game to destress and have fun, while also not needing to worry about how playing the game is going to impact future workflow.

The game I am planning to make is a version of snake, a simple computer game from 1997, in which the player controls a snake that moves around a grid and collects "food" to gain points and grow longer, while trying not to hit any obstacles (including its own tail). As this is a concept that has been done multiple times already, I plan on changing the idea a little bit, allowing the player to compete in the game against other players in real-time, with the winner being the last "snake" standing.

My End Users

As I am developing a video game, I am ideally looking to attract as wide a range of people as I can to my project. However, preferably the stakeholders will represent 14–18-year-olds. This in itself is still a wide range of end users, as some people in this age range are only looking for casual gaming and a way to destress and pass the time, while others in this age range are able to sink in hours of their time into mastering the mechanics behind a game. Therefore, I am intending for this game to be easy to pick up and learn, as well as fun to play casually, while also providing a small sense of competition that some players enjoy experiencing.

As my main demographic is young people, I will have to make sure that my User Interface is intuitive to understand and easy to navigate. This is to ensure players do not spend the majority of their time trying to move through multiple unnecessary menus, get frustrated at how hard it is to get into a match, and then not play a match in the first place.

In addition, as these young people may not have much time in their schedules to play my game, I have to ensure that the duration of a single match is fairly short (between 5 and 10 minutes depending on how well people play). This allows for a balance between not spending too much time on the game such that other tasks are left unattended, and not spending too little time such that each match seems like a waste of time completely.

Existing Solutions

Case Study 1:

Like I mentioned above, my project will be based off of the game *Snake*. This game consists of a small grid (which may or may not contain walls depending on the difficulty) and – initially – 2 blocks – one denoting the player's snake at its initial size, and the other denoting the "food" the player is meant to collect. The player controls the snake using 4 buttons (one for each direction), and every time the snake collects a piece of food by directing its head to the food block, it grows in size by 1 block. The snake must also avoid hitting any obstacles including its own tail, thus making the game more difficult as you increase your score.

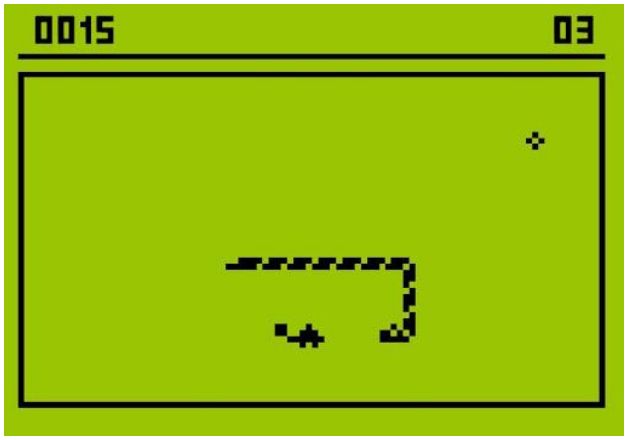
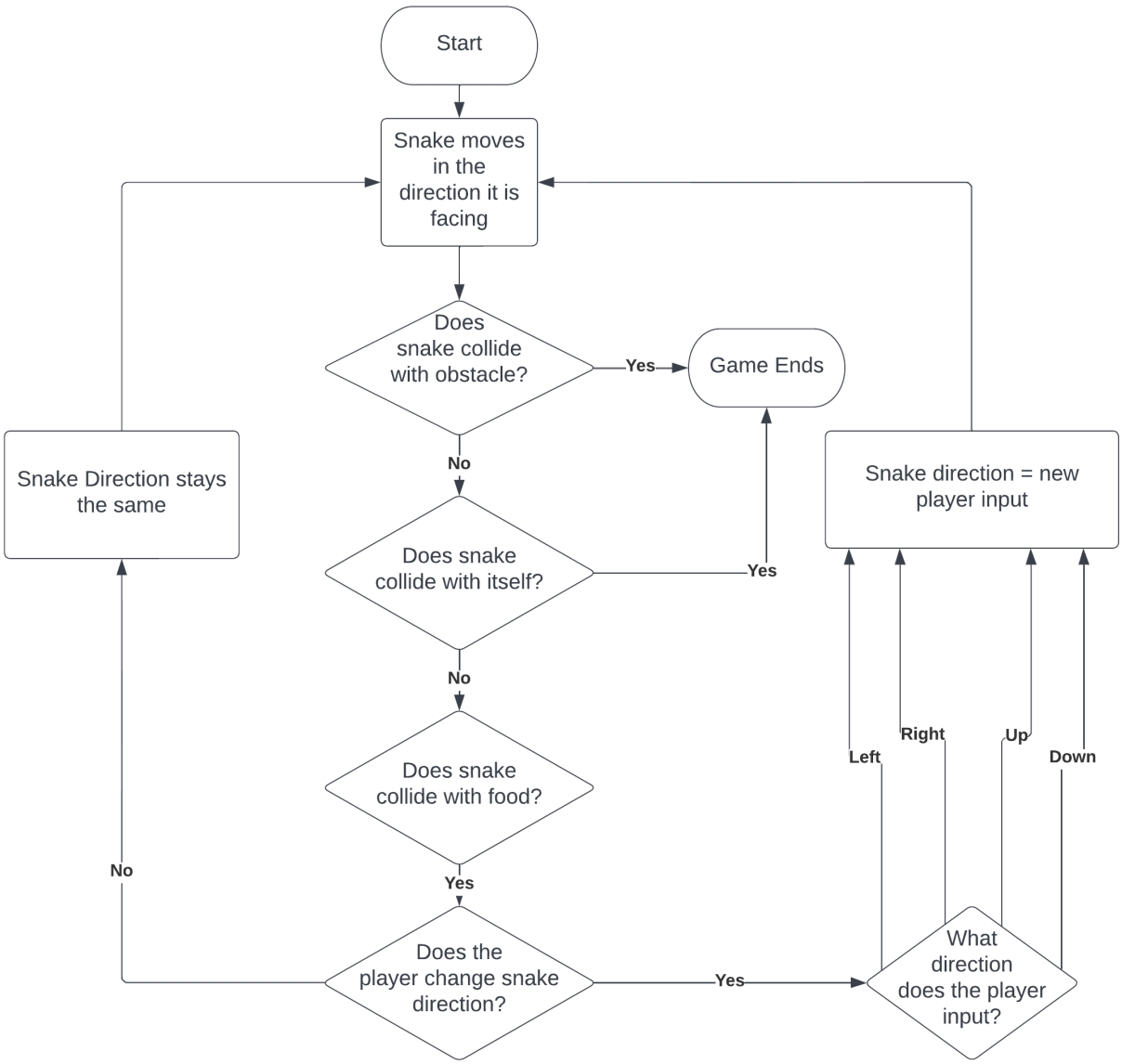


Figure 1: The original game of Snake on the Nokia 6110 in 1997. Its sequel, Snake II, included level progression and the ability to go through

Here is a simple flowchart model on how the original snake game functioned:



Although this game on paper seems like a fun idea, by itself this design has been carried out numerous times, and the gameplay itself can get fairly tedious after a while. Simply making another snake game would not be very effective as people can just choose to play a different game completely.

Case Study 2:

One of the games that takes an initially simple game and modifies it to become more exciting is *Tetris 99*. This take on the traditional *Tetris* game turns it into a "Battle Royale", where the aim of the player is to be the last one standing. Here, whenever a player clears 2 or more row of blocks, they send a "garbage" row of blocks to another player. That player gets a time delay before the row(s) of blocks appears at the bottom of the screen. The more rows of blocks you clear, the more "garbage" rows of blocks you send to other players. As with normal Tetris, if you reach the top of your screen, you are eliminated from the game.

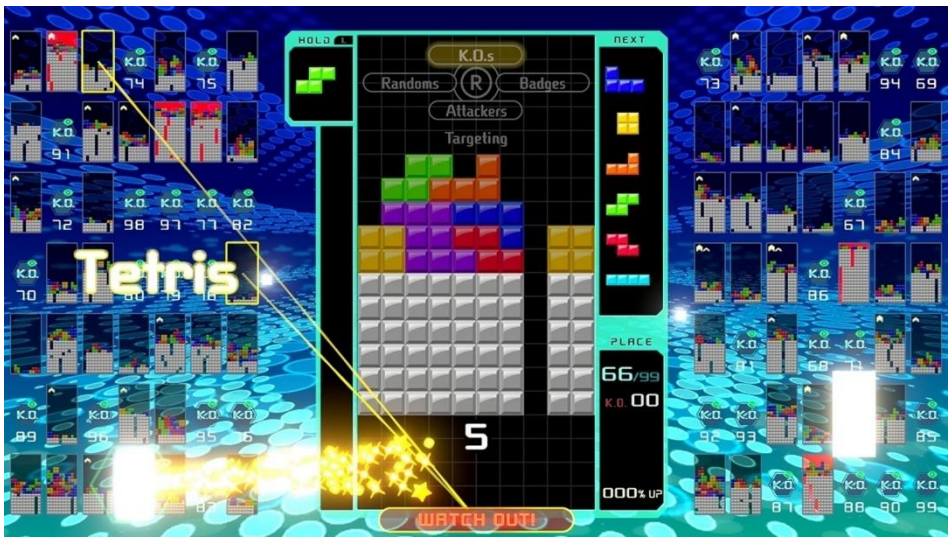


Figure 2: Gameplay of Tetris 99 (players can target "garbage" rows to opponents from the interface at the top of their screen, changing who they attack based on number of kills, who is attacking the player, or completely at random).

Case Study 3:

Another game that does a similar thing to Tetris 99 is *Super Mario Bros. 35*. Much like *Tetris 99*, the player is put in a normal game of *Super Mario Bros.* against 34 other people, each player having a time limit of 400 seconds maximum. Every enemy defeated in a person's game sent that enemy to a targeted opponent (in the same way that *Tetris 99* handles attacking), as well as adding time to the player's own game. A player can also obtain items with the coins they collect from inside each match to aid them in defeating enemies. A player is defeated when they either die to an enemy or run out of time, and the last player standing wins.

Figure 3: Gameplay of Mario 35

Looking through how these games worked, I extracted some features that I could carry over to my project to improve the overall quality of it to my stakeholders. First of all, both games are not very graphically intensive compared to what is seen in other titles today, so it may not be necessary to use high level graphics to make the game play better. In addition, the two games above do not add too much to the base game, only opting for players to do well in each match by playing the game as they would do if there was no "battle royale" aspect. Overcomplicating features on top of the foundations of Snake would make the game not fun for some people, and so I should try and keep extra features I add into my project as simple as I can. Finally, both games allow the player to view

how other players are doing in real time. Although this may be more of a quality-of-life feature and an afterthought, this could help give a visual indication of how the player is doing, increasing the sense of accomplishment the player feels if they are doing well. However, I could also see that in the middle of a match, the games' UI would feel quite cluttered and chaotic (especially in *Tetris 99*), which could end up being very distracting for the end user. To reduce this, I might use a simpler colour scheme for my interface (for example, using a single black background instead of a moving background and fewer visual effects would help reduce the level of distractions the user may have).

Limitations

There are obviously going to be some limitations with the project's proposal. One of these might be the inability for those with visual impairment to navigate the UI and play the game. To help try and reduce this inability as much as I can, I will not use any complex shapes in my interface, and I will use distinct block colours to help distinguish various parts of the game, making it much easier for those who struggle seeing to enjoy the game.

Another limitation could be how some people may not have as powerful devices as others to play this game (for example, some users may have laptops that only use IGPUs integrated into their CPUs). In order to include as many people as I can, I will refrain from creating graphically complex objects and using too many visual effects.

One more limitation of this project is the range of devices that can access this game. Users have a lot of devices that have the ability to play video games, with the majority of people using their phones (shown by most of the revenue in the video game market coming from the mobile gaming sector). While creating this project for mobile devices allows it to reach the widest audience possible, it does cost money to publish an application on major mobile app stores. In the Apple App Store, an annual fee of $99/£81 is needed to keep an app on their marketplace, and in the Google Play Store, only a one-time purchase of $25/£21 is required. While these may seem like fairly cheap options, spending no money at all is preferable, and thus developing the game for desktop devices would be ideal.

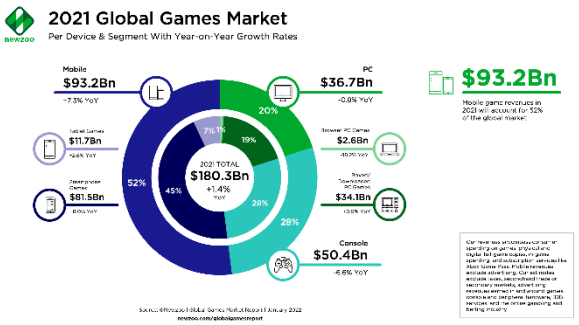


Figure 4: Shares of global revenue in the Video Game Industry based on device in 2021. Notice how the mobile gaming market earns more than the other games markets combined.

Research Plan

I will intend on conducting a series of interviews on a few people in the 14-18 age range to gather a representative idea on what the stakeholders are looking for in my project. I will be asking questions specifically about their favourite game, features that they like about their favourite games, whether or not they have experienced different genres of games as well as their opinion on those genres.

Justification of Computational Methods

Decomposition:

The main project can be divided into multiple problems, such as joining a match, controlling the snake, and managing scores between players. Each of these problems can be further divided into more steps. For example, controlling the snake may be divided into the following steps:

1. Check the current direction of the snake
2. Move the snake 1 unit forward in the current direction
3. Check if the snake collides with a wall or itself
4. Check if the player has made a new directional input
5. Change the direction of the snake accordingly

At its core, at each frame the game performs these tasks in the order specified, and it does so at fast enough frame rates that each input the user makes results in an immediate response. By dividing each problem the game is trying to solve into multiple smaller problems, doing so again and again until each problem can be solved with a single function, the problem reduces in complexity, reducing the chances of overcomplicated programs being coded in, which could reduce overall game performance for users, as well as making it easier to identify and thus fix any errors that may come up during development.

Divide and Conquer:

Following on from the Decomposition computing method, each problem, once being halved through multiple iteration until at its smallest subproblem, can be easily solved. Each subproblem can be then merged into solved problems, which can be further merged until a full solution to the original problem has been reached. This, too, improves the efficiency of both coding and testing any programs during development.

Abstraction:

Abstraction is the process of removing excessive details of a problem, only leaving the relevant details, thus simplifying it. Abstracting details off problems helps with development largely, as effort does not need to be made to solve problems that don't matter at the time, which reduces overall development time.

For example, in terms of controlling the snake, only a maximum of 6 buttons are going to be used (4 buttons for directional movement, as well as 2 buttons for things like power-ups that may be implemented). Obviously, a typical keyboard has more than 6 buttons, and the rest of the buttons that will not be used will be abstracted out, removing the need to think about the implementation of those buttons for the movement of the snake.

Another example of this could be a leaderboard that may be implemented into the game to keep track of player scores. Here, the user doesn't need to see, say, how many "food" blocks other players' snakes have collected, or how long their snakes got before they lost. All the player needs to know about other players' performances is their final score, so unnecessary details such as final snake length do not need to be accounted for when developing a leaderboard.

Project Investigation

Questionnaire

I will now outline the questions I will ask each stakeholder, finding out what they liked, disliked, and desired from games they may have played in the past.

Form Questions:

1. Have you ever played a version of "Snake" before?
2. If yes:
    a. If so, what feature(s) of the version of this game you played do you like?
    b. What feature(s) did you dislike about this version? If so, what feature(s) of the version of this game you played do you like?
    c. Are there any novel features that you would like to see in a new version of "Snake"?
3. Have you played or seen gameplay of games such as "Tetris 99" or "Mario 34"?
4. If yes:
    a. If so, what feature(s) of these games do you like?
    b. What feature(s) of these games do you dislike?
    c. What feature(s) did you feel were missing that you would have liked to see in this game?
5. If no:
    a. If not, here is gameplay from "Tetris 99" (video). Based on this gameplay alone, what feature(s) (visually, audibly, or otherwise) did you like?
    b. What feature(s) did you not like?
    c. What feature(s) did you feel were missing that you would have liked to see in this game?
6. What resolution do you typically play at?
7. Any other comments you would like to make?

Question 1 establishes the stakeholder's history with the game of *Snake* and asks if the stakeholder has played a version of the game before. This is important as it is necessary to know if their opinions are based on previous experience or simply speculation.

Should the stakeholder have any experience with *Snake*, question 2 delves further, asking them on their opinions on their played version of *Snake*. This question is divided into 3 parts, asking what they liked and disliked about their version, as well as what they would have liked to see should a new version be made. This helps understand what stakeholders already like and dislike about the current system (and thus what should be kept or removed) and what additions on top of the current system I should make to further improve the experience to the stakeholders.

Question 3 follows on by asking the stakeholder if they've heard of the other two existing solutions stated in the *Existing Solutions* section: Tetris 99 and Mario 34. Like Question 1, this question establishes the stakeholder's history with either of the two games.

If the stakeholder has played either of the two games, Question 4 acts the same way as Question 2 does, being divided into 3 parts asking for what the stakeholder liked and disliked in either game. The use here is to pick out features that work well in both *Tetris 99* and *Mario 34* and implement them in this project.

If the stakeholder hasn't played either of the two games, Question 5 shows the stakeholder a video of gameplay of *Tetris 99* and asks them, purely based on what they saw/heard, what they liked and

disliked about this game. This focuses purely on first impressions, and thus shows what attracts new users to a game like this, which could then be used in the project.

Question 6 asks the stakeholder what resolution they typically play at. The results influence the decision on what resolution the game should default at, as well as if a resolution changer must be added into the game.

The questions conclude with Question 7, asking if stakeholders have anything to say that may have been missed.

Form Results:

1. **Have you ever played a version of "Snake" before?**
   "Yes." [7]
   "No." [1]

2. **If so, what feature(s) of the version of this game you played do you like?**
   "The building challenge, the change to your own character (the snake) being the obstacle to overcome, the tracking of progress and displaying of stats to gauge progress."
   "Score counter, power ups (for example a helmet that when you collided with yourself resets you at the beginning but with all your body parts), visually appealing graphics, simple but good music, low latency."
   "The game has a massive play area"
   "Simple gameplay, and thus easy to get high scores"
   "Smooth and responsive controls"

3. **What feature(s) did you dislike about this version? If so, what feature(s) of the version of this game you played do you like?**
   "It's formulaic, becoming boring and repetitive quite quickly, there's little incentive to continue playing after a few times of playing, the designs are rather generic and not particularly interesting to look at, it's rather limited, there are set ways to win with little choice in a path if you wish to get as far as possible."
   "It's still snake in the end so it's a bit boring you know?"
   "Gets boring fast, no innovation in gameplay, no multiplayer"
   "Not very interesting gameplay, Lack of complexity"
   "I didn't like the simplicity and it got boring really quick"

4. **Are there any novel features that you would like to see in a new version of "Snake"?**
   "Customisation, different sized maps, some kind of incentive in the form of a progressions system perhaps or tied into the customisation aspect, perhaps create achievements for certain milestones."
   "I want to see customisable skins and ability to send emotes like in clash royale, also u need a ton of particles and sound effects when someone dies so that it's given a modern feel which builds on top of the retro nostalgia like Tetris 99"
   "arcade mode like in fruit ninja with a lot of power ups and things that make the snake longer"
   "Multiplayer, 3d, good graphics"
   "Powerups like Teleporter, disjointed snakes, control two at once, mirror"

5. **Have you played or seen gameplay of games such as "Tetris 99" or "Mario 34"?**

"Yes." [6]
"No." [2]

6. **If so, what feature(s) of these games do you like? Separate multiple features with a comma.**
   "The vivid imagery, the slowly rising challenge, the multiple ways to deal with said challenge and the pressure the player faces often being the primary obstacle for the player's success until later rounds."
   "The multiplayer aspect, since it's based off an original game that's good the gameplay is good, good music, good visual effects"
   "I love the sound effects which really enhance your gameplay experience when u play and makes u feel like u are popping off when u eliminate someone"
   "Fast pace, high skill ceiling"
   "Hectic gameplay, multiplayer, responsive inputs and gameplay"

7. **What feature(s) of these games do you dislike? Separate multiple features with a comma.**
   "It's rather typical, repetitive, boring, there's a lack of progression or incentive to continue playing."
   "Lag, Nintendo's net code, joycon drift, bugs such as Tetris misdrops."
   "no matchmaking based on skill level"
   "Hard to win, repetitive, not much other functionality"

8. **If not, here is gameplay from "Tetris 99". Based on this gameplay alone, what feature(s) (visually, audibly, or otherwise) did you like?**
   "music, sfx, colour, aesthetic, infographics"
   "Looks very different from traditional Tetris. Gameplay looks exciting with a new twist. Graphics flashy and cool."

9. **What feature(s) did you not like?**
   "nothing"
   "Looks very confusing (may require tutorial), no idea what's going on"

10. **What feature(s) did you feel were missing that you would have liked to see in this game?**
    "Progression, customisation, different blocks to choose from, different variations of the basic Tetris formula, different materials for Tetris blocks perhaps (soft body materials)?"
    "a variety of modes"
    "More minigames, more gimmicks"

11. **What resolution do you typically play at?**
    "1920 x 1080" [4]
    "1280 x 720" [2]
    "2560 x 1440" [1]

Questionnaire analysis

Although the sample size of the survey was low, there is still important information that could be extracted for this project. From the first question, I can gather that the majority of the chosen stakeholders are in some level of understanding of the game *Snake*, and thus can help me to improve the end product. The overall perception of *Snake* was that while the game had a simplistic

goal and set of controls, and it had a decently increasing difficulty as the snake grows in size, the simplicity of the game may have meant that the game became increasingly more boring over time. With the same things to do again and again until the player loses, after some time, the gameplay does get very repetitive and tedious, and there would be little reason to continue playing the game, just like what I stated when I first talked about *Snake*. Unless some extra features were added in (some features stated by answers in the questionnaire included a set of powerups the snake could obtain, different sized/shaped maps, a levelling and progression system, and possibly cosmetics like skins) that provide an incentive for users to keep playing this game, the overall effect of it would be minimal.

Surprisingly, a higher proportion of stakeholders had heard of either *Tetris 99* or *Mario 34* than I had initially thought. In addition, my initial assumption of graphics not being a very important factor in this project was proven to be false. The majority of those who had already seen/played *Tetris 99* or *Mario 34*, as well as those who saw the provided gameplay of *Tetris 99* agreed that one of the positive factors of these games was their flashy visuals (going forward with this, a possible method to still cater for those who may be visually impaired or may just want lower-end graphics is to add a button that reduces the quality of or removes any flashy visual effects). In addition, sound effects like background music were a highly appreciated feature of these games by the stakeholders, as it adds to the overall feeling of intensity in each match (for example, the music speeds up and gets more intense the less players there are left in a match). The “fast-paced, high skill ceiling” gameplay in these matches makes them more unique and exciting, reducing the effect of what, typically, is a repetitive game. On the other hand, for those who already played *Tetris 99* especially, some concerns raised consisted of a lack of progression in-game to keep playing, issues with networking and in-game bugs occurring mid-match, and a lack of skill-based matchmaking, resulting in players who want to play simply to have fun getting beaten very quickly by more competitive players, which does not solve the main problem at all. For those who have never seen either game, a concern that was raised was that the game looks very confusing and chaotic, which may put off some people from playing the game after a few matches. However, this could be easily fixed with some sort of tutorial provided when the user first starts up the game. When asked on what people could add to *Tetris 99* or *Mario 34*, three main features specified were a progression system (like with *Snake*), cosmetic customizability, and possibly more modes to keep users playing the game if they get bored with the main match mode.

When asked about the resolutions they played at, the majority of stakeholders stated that they play at 1920 x 1080, with fewer saying they play at 1280 x 720 and 2560 x 1440. From this, while the default resolution for the game would be 1080p, a resolution changer may still have to be implemented in order to cater for those playing with different sized screens.

Requirements

Development Requirements

I will be using the Unity IDE to develop this game, with the program being written in C#. I chose Unity, not only due to its ability to implement fairly complex objects into a game (unlike Pygame, where games typically look less visually appealing), but also due to its ability to handle networking well, as well as its simple drag-and-drop feature, simplifying the development process massively.

Hardware and Software Requirements

Hardware:

The hardware requirements for this game will not be too restrictive, as the game does not need hugely complex environments of photorealistic graphics that require expensive graphics cards or CPUs to process, so a wide range of hardware combinations would be compatible and suitable for running this game.

Hardware Requirement	Justification
Intel Pentium 4 (2.8 GHz) and above or Athlon 64 X2 3000+ (2.0GHz) and above	A CPU that supports the SSE2 instruction set is needed to run most games
Nvidia GeForce 6800 or RADEON X800 and above	A Graphics Card that supports DirectX 9 (shader model 3.0) or DirectX 11 with feature level 9.3 capabilities, with at least 256MB VRAM is needed to run most games
100-200 MB storage	Typical storage space needed for a project of this size to run on a user's computer

Software:

The Operating System used for playing this game would need to be either Windows, Linux or MacOS, as these are the only 3 OSs supported games made with the Unity engine, using the Unity Player. Other than that, as the end product will be an executable file, no other software is needed (unless I publish the game on Steam, in which case the Steam Client would also be necessary).

Software Requirement	Justification
Windows XP or above	OS required to run Unity Player
MacOS 10.12+	OS required to run Unity Player
Ubuntu 20.04, Ubuntu 18.04, and CentOS 7	OS required to run Unity Player

User Requirements

Design

Requirement	Explanation
Visually striking graphics and flashy effects	Helps improve the overall aesthetic of the game, giving the user a feeling of excitement and thus enjoyment.
Easy User Navigation	Improves general user quality of life, allowing the user to spend more time playing matches instead of trying to navigate menus to find a match.
Interface to notify players of their placement in a match, as well as how many eliminations they get	Helps notify players on how well they are doing, and encourages them to keep playing, especially if they're already doing well

Satisfying sound effects	Much like flashy effects, they help improve the aesthetic of the game, offering satisfaction to the player when they are doing well, as well as increasing tension as the player gets further into a match. Specific sound cues may also help the player in understanding the game

Functionality

Requirement	Explanation
Smooth and Responsive Controls	A system that can quickly and easily react to inputs given by the user improves gameplay. Less responsive controls mean an overall worse experience with many hitches beyond the user's control, reducing satisfaction, and is thus something I want to avoid.
In-Game Progression System	Allows the user to track their overall progress and provides an incentive to play more and get better in order to get a higher level, for those who may want to invest extra time into their game.
Matchmaking	Allows for players to connect to public matches and play against each other in real time. This added multiplayer feature makes the game a lot more interesting than simply single-player *Snake*, thus improving player engagement.
Tutorial	A short game teaching players how to play the game when they first play, helping them understand the mechanics of the game better.
Resolution changer	Helps players to change their game's display resolution, either to help the game fit screens better, or to increase frame rates.
Achievements/Missions	Specific tasks that the player can work towards and complete in order to receive certain rewards, thus proving a reason to do better in the game.
Customisation Items (Optional)	Items such as skins that can be equipped in a dedicated menu that can customise the look of a player's game. Can be unlocked from achievements and doing well in games, providing a further incentive to playing.
Power-Ups	Special abilities that improve a player's chance of winning or makes an opponent's chance of winning worse. Provides a twist to the general *Snake* "formula", making the game less boring and thus increases player engagement.
Minigames (Optional)	Special modes that are included with the main gamemode in this project, designed to add versatility to a game, so that if players get bored with one gamemode, they can simply play another, retaining player engagement.

Success Criteria

Criteria	How to evidence
An easy-to-navigate Interface	Video of Menus with clearly labelled buttons and simple navigation.
A way to allow the user to configure in-game sound	Video of a Volume Settings Menu
A way to allow the user to configure graphics such as resolution	Video of a Graphics Settings Menu
A menu that allows the user to customise the looks of the snake, food and the background	Video of customisation menu
Progress Tracking with a Progression System	Video of a menu showing user wins, losses, eliminations, power-ups used throughout all games played, as well as what level they are and how far they are until the next level
A way to enter a matchmaking lobby	Video of Multiplayer Matchmaking Menus
A way to spawn players in separately	Video of starting the game
Real-time multiplayer networking	Video of visibly syncing between the local player and all other players
A way to change the direction of the snake using user input	Video of inputting directional keys to change direction of snake
Detection for when a snake collects a food item and for the score of the user to increase accordingly	Video of snake running into food, length and score increase
Sound/Visual effects when the Snake obtains food	Video of effects playing when player gets food
Detection for when the snake collects a certain number of food and for the speed of the snake to increase accordingly	Video of snake having a score which is a multiple of 5, and increasing speed
A way to target different players	Video of player inputting a key and changing type of targeting
A way to obtain powerups using user input	Video of player inputting a key and randomly rolling a power-up
Sound/Visual effects when the Snake obtains a powerup	Video of effects playing when player gets powerup
A way to send the effects of a powerup to other “alive” players in game	Video of player inputting a key and sending an equipped powerup to the targeted player or themselves
Sound/Visual effects when the Snake uses a powerup	Video of effects playing when player uses powerup
Detection for when the snake hits an obstacle and for appropriate loss visual/sound effects to appear	Video of snake running into food, death effects play
A way to notify players when one player dies	Video of death effects shown to other players on death
A way to notify players when one player wins	Video of win effects shown to other players on win
A way to notify players when they get a kill	Video of Kill Count of player increasing when they get a kill
A way to notify the user of their in-match position placement, as well as how many eliminations they get	Image of screen showing these values to the player when the game ends

A way to provide new users with a tutorial on how to play the game	Image of screen explaining how the game works

Section 2: Design

Design Objectives:

I will now divide my success criteria into input, processing, and output considerations. I will also add in aesthetic consideration based on what I want the application to look like.

Aesthetic considerations

- The Screen size will default to 1920x1080 pixels, but can change according to the user's preferences, as different people will have different monitor sizes to play on.
- To keep up with user requirements of :
    - The colour of the snake will default to green
    - The colour of the food will default to red
    - The colour of the background will default to blue, and will be animated. The user can opt to remove these animations from the settings menu
    - Food, upon collection, will produce an effect to the user.
- The main menu will feature the title of the game, as well as the play, options and Exit buttons
    - Each corresponding menu would have its own set of buttons and dropdown menus to suit their respective functions.
    - Buttons that are present in the game will either have a slight colour change or opacity change when highlighted and pressed.
    - Any pop up menus can appear with a "pop" type animation

Input Considerations

- A way to enter a matchmaking lobby
- A way to change the direction of the snake using user input
- A way to obtain powerups using user input
- A way to allow the user to configure in-game sound
- A way to allow the user to configure graphics - such as resolution - and visual effects
- A way to allow the user to configure controls
- A menu that allows the user to customise the looks of the snake, food and the background

Processing Considerations

- Real-time multiplayer networking In game
- Progress Tracking with a Progression System
- Detection for when a snake collects a food item and for the score of the user to increase accordingly
- Detection for when the snake collects a certain number of food and for the speed of the snake to increase accordingly
- A way to send the effects of a powerup to other "alive" players in game
- Detection for when the snake hits an obstacle and for appropriate loss visual/sound effects to appear

Output Considerations

- An easy-to-navigate Interface
- A way to access Missions/Achievements

- Sound/Visual effects when the Snake obtains food
- Sound/Visual effects when the Snake obtains/uses a powerup
- A way to notify the user of their in-match position placement, as well as how many eliminations they get
- A way to provide new users with a tutorial on how to play the game

Problem Decomposition

The overall problem is a large one to solve, and attempting to solve it in one go would be very difficult due to a lack of a path that needs to be taken in order to find a solution. Thus, a decomposition diagram can be used to divide each problem into a set of sub problems, represented as self-contained modules.

This results in many benefits, especially with testing of code. If all sub-problems are self-contained, then changes in one sub-problem will not affect another. If there is any issue in the testing of code, I can easily follow the decomposition diagram to find out where the issue lies, making tracking down and fixing any bugs much easier and faster.

In addition, by using modules, I can reuse parts of each module to apply to other sections of my project, reducing time taken to write code (for example, reusing code used to implement each power-up in the game).

Thus, I will split my problem into a series of sub-problems which, put together, will represent the larger problem as a whole. Once I have done that, I will continue to divide these sub-problems until they become as small as they can be.

Menus:

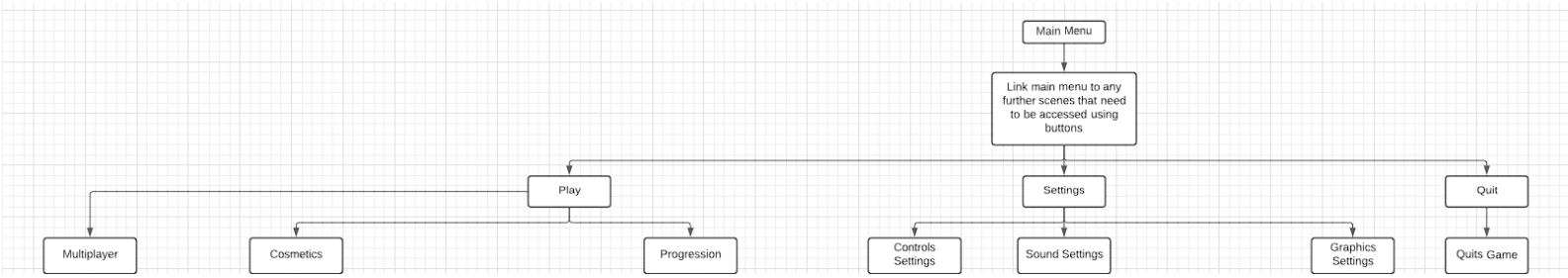


Figure 5: Decomposition diagram for Menus

The menu subproblem will encompass everything to do with the user experience of navigating the menus that would lead the user into a match. By breaking down this subproblem into even smaller problems, I am able to designate a path I can follow to get all of these sections done, making the development process faster and easier.

Firstly, I had the Play subproblem, which I further decomposed into Multiplayer (allowing the player to enter lobbies with other players to play), Cosmetics (allowing the player to both view and equip any skins/backgrounds they own) and Progression (allowing the player to view their achievements and current level), all of which have further decomposition as shown in the images below (Multiplayer will have even further decomposition shown later):

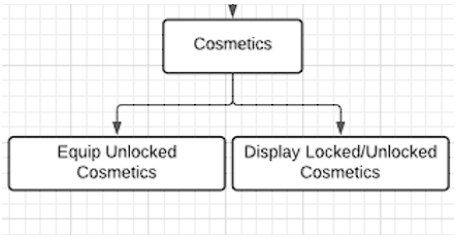


Figure 6: Decomposition Diagram for the Cosmetics Menu

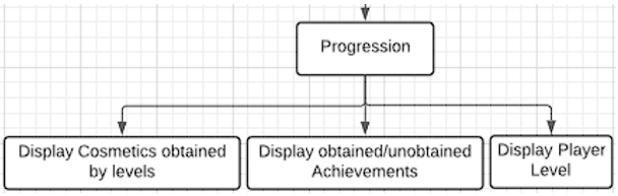


Figure 7: Decomposition Diagram for the Progression menu

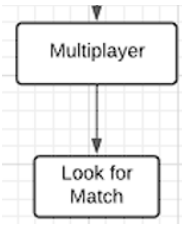


Figure 8: Decomposition Diagram for the Multiplayer Menu

The settings subproblem has also been divided into Controls, Sound and Graphics settings, which all have their own place in the main settings menu and have the following decompositions:

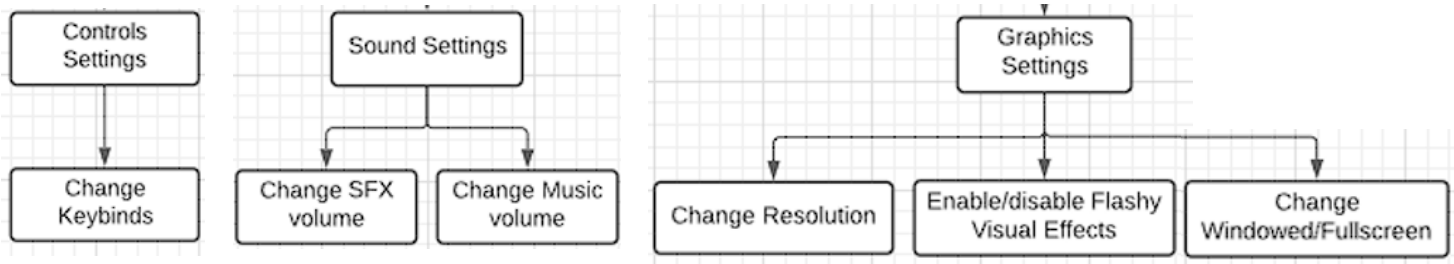


Figure 9: Decomposition Diagram for the Settings Menus

Finally, I also broke down the menu subproblem into Quitting the game, which requires no further decomposition as it is a very trivial subproblem to implement.

Gameplay:

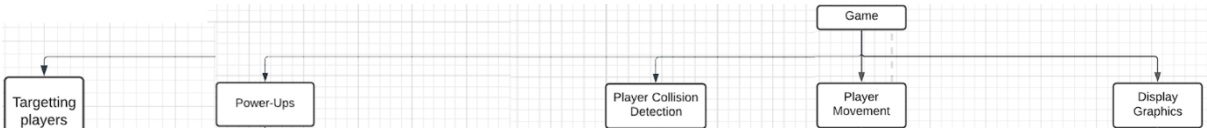


Figure 10: Decomposition Diagram for in-match Gameplay

The Gameplay subproblem of this project will cover how the game looks and is played while in a match. I have also divided this problem up into further subproblems in order to, like above, designate a path to easily complete the main subproblem. In addition, I added details on the basic functionality of each subproblem to make it easier to understand what my code is meant to do upon completing each subproblem, which will help a lot when testing code.

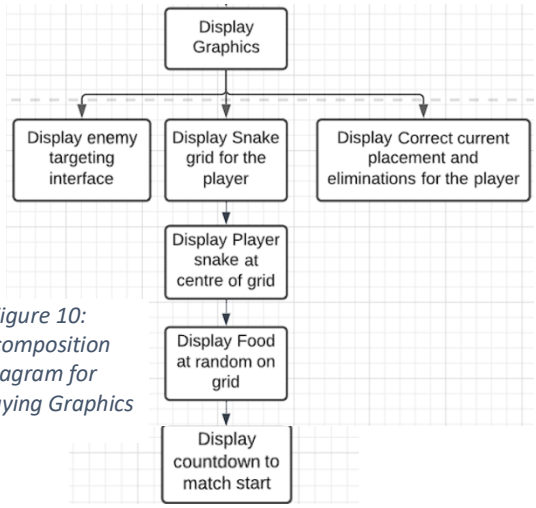


Figure 10: Decomposition Diagram for Displaying Graphics

One of these smaller subproblems is Displaying graphics, which I have broken down into 1) Displaying the snake grid, snake, food (which position is updated in real time) and a countdown when the match starts, 2) displaying the targeting interface for the user, as well as displaying sufficiently up-to-date placement and elimination counts for the user. These tasks will be fairly trivial, as it mainly consists of assigning existing variables to UI components (apart from the targeting interface, which would need only a little bit more work), and thus will not receive further decomposition:

Another one of these subproblems is Player Movement, Which constantly moves the player's snake one unit per set time period, as well as constantly checking for any directional inputs from the user, and changing the direction of the snake accordingly. Player Movement also ensures that the speed of the player's snake increases with every 5 food consumed. Decomposition is done here in order to reduce complexity in doing a larger problem all at once by dividing it into smaller, more manageable problems:

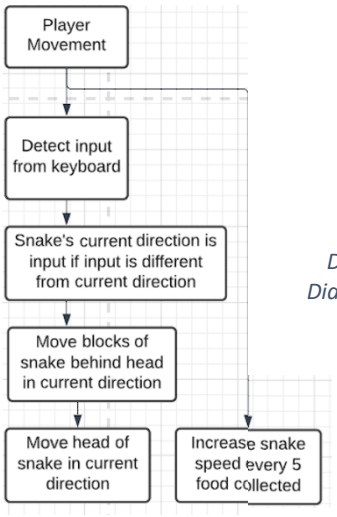


Figure 10: Decomposition Diagram for Player Movement

One more of these subproblems is Player Collision Detection, which reacts differently based on whether the player's snake collided with food, walls, or itself. The problem is decomposed here because the same events occur when the snake collides with itself or the walls, and so decomposition can allow for reusability of the same code for both circumstances, making the development of the project easier:

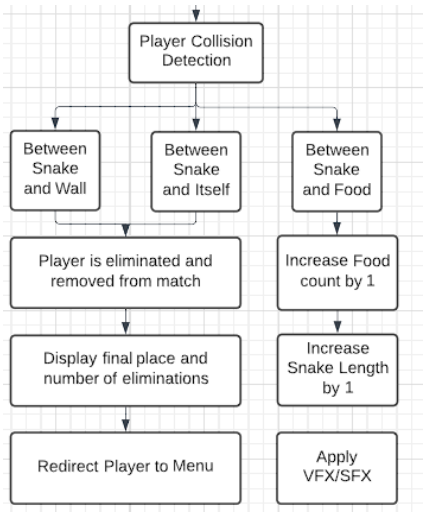
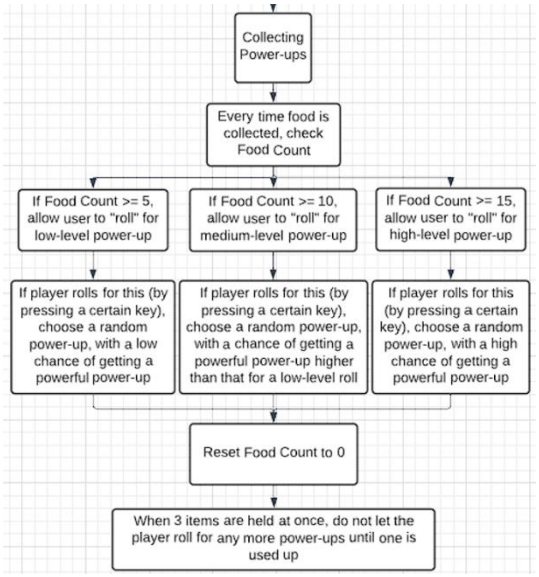


Figure 11: Decomposition Diagram for Player Collision Detection



The final subproblem here is Power-ups, which in itself is divided into 3: Collecting Power-Ups, Using Power-Ups and Player Targeting.

Collecting Power-Ups relates to checking if a player's current Food Count is past a certain threshold, and then if so, unlocking the ability to "roll" for a random Power-Up, with rarities scaling on how high the Food Count is. Once the player rolls for a Power-Up, their Food Count resets, and the player needs to collect more food to roll again until they hit their 3 Power-Up limit.

Figure 12: Decomposition Diagram for Collecting Power-Ups

Using Power-Ups relates to checking if a player is currently holding a Power-Up, and if so, applying the effect on the Power-Up on either themselves or opponents, depending on the Power-Up. This also applies elimination score to players and removes the Power-Up from the player when it has been used:

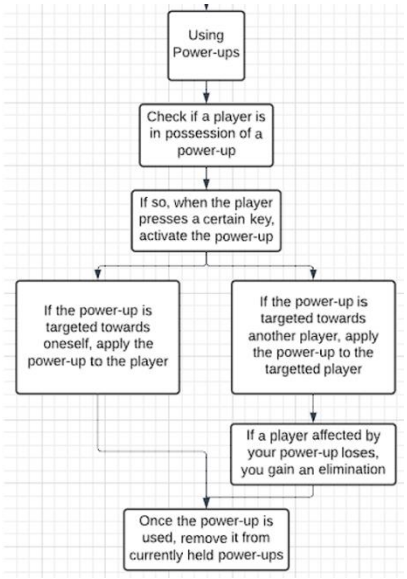


Figure 13: Decomposition Diagram for Using Power-Ups

Targeting players relates to how Power-Ups affecting opponents are distributed. Opponents can be targeted in 4 different ways (shown in Figure 14) and targeting modes can be changed by the press of a key:

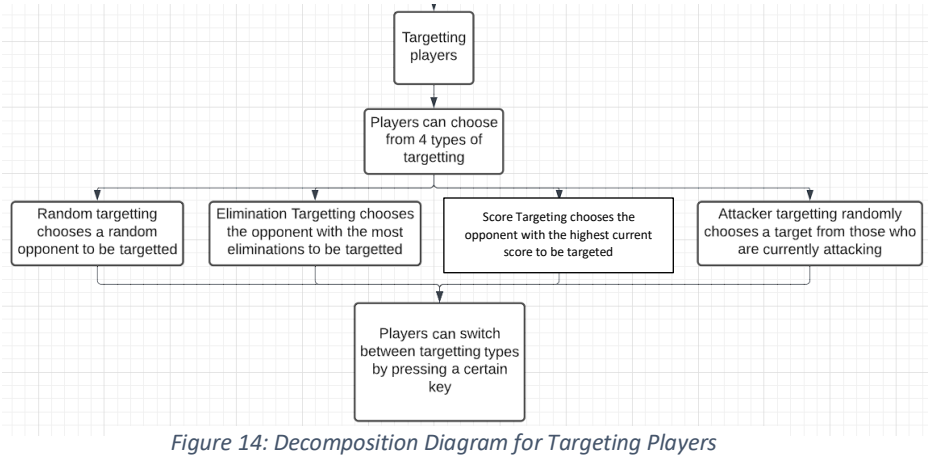


Figure 14: Decomposition Diagram for Targeting Players

User Interfaces:

Main Menu:

Figure 15: Design for Main Menu

Settings Menu:

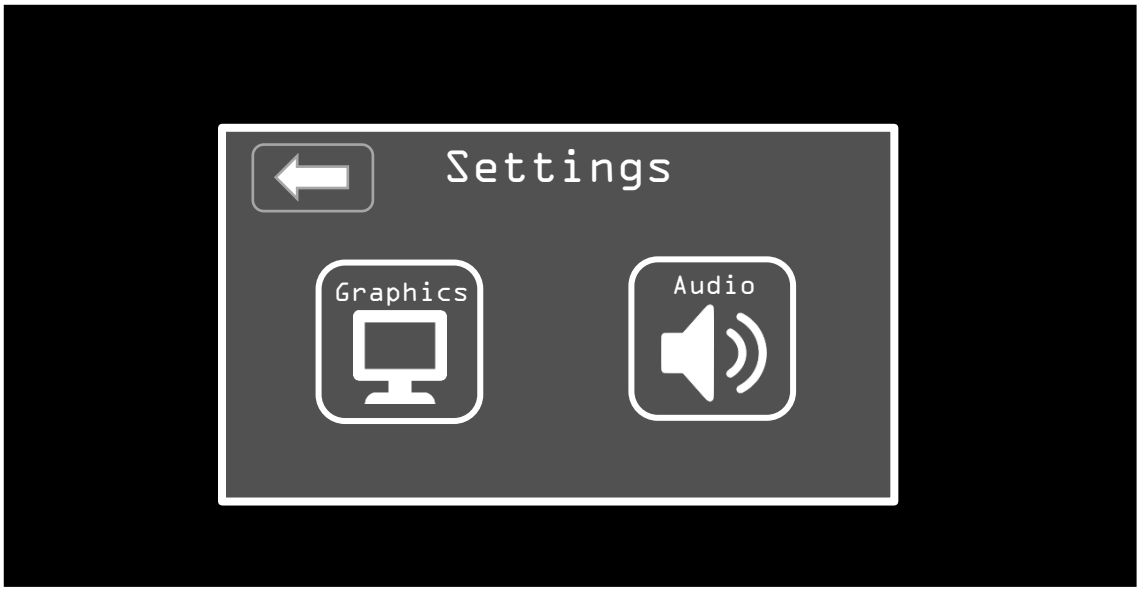


Figure 16: Design for Settings Menu

Controls/Graphics/Sound Menu:

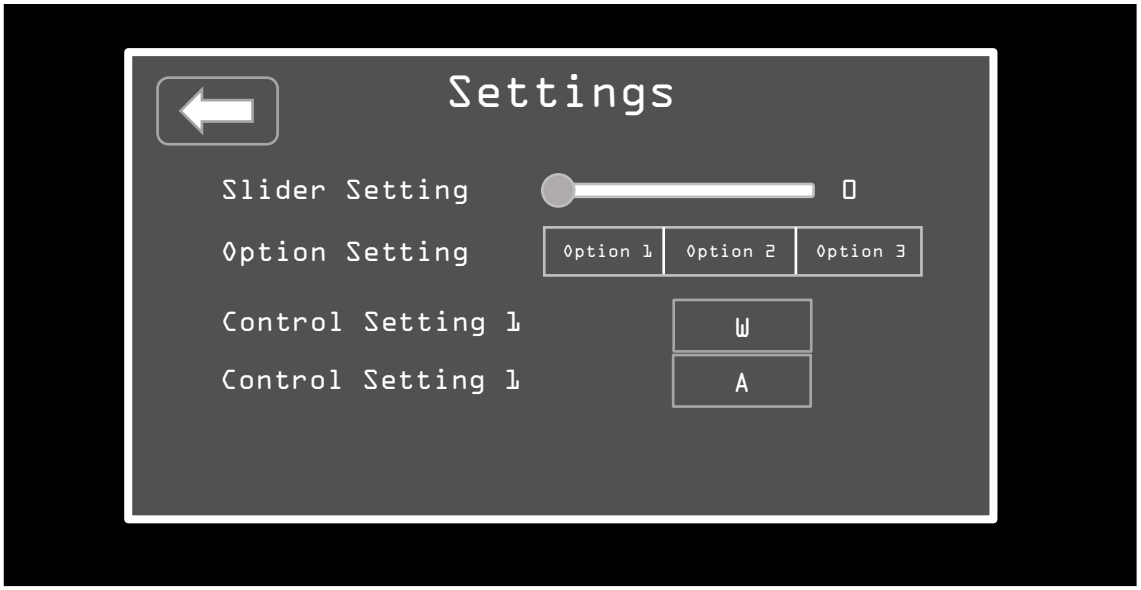


Figure 17: Design for Extra Settings Menus

Play Menu:

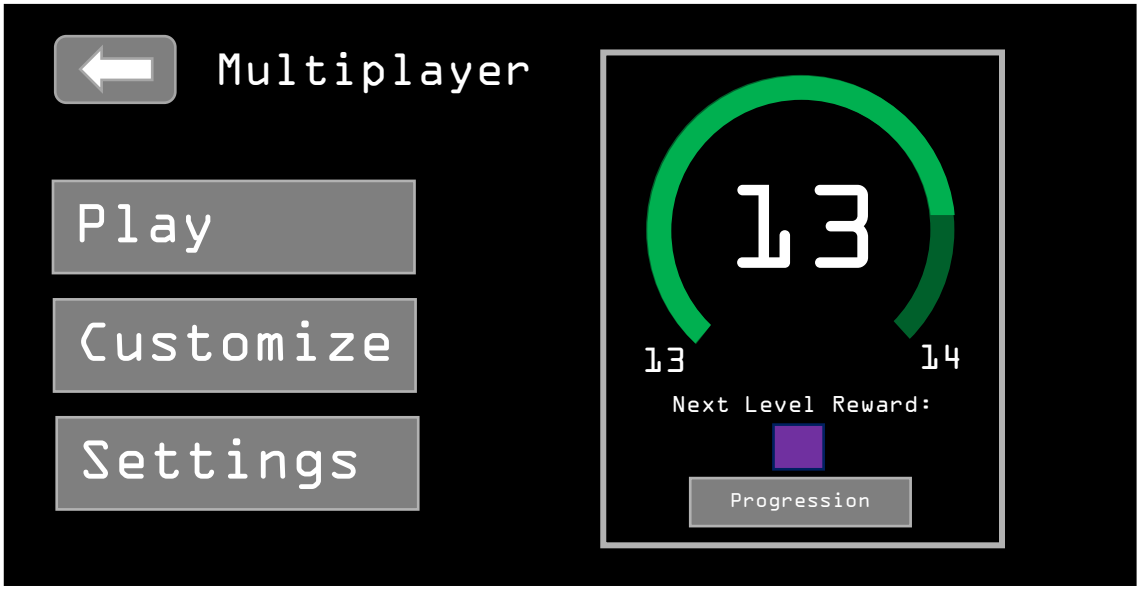


Figure 18: Design for Multiplayer Menu

Customization menu:

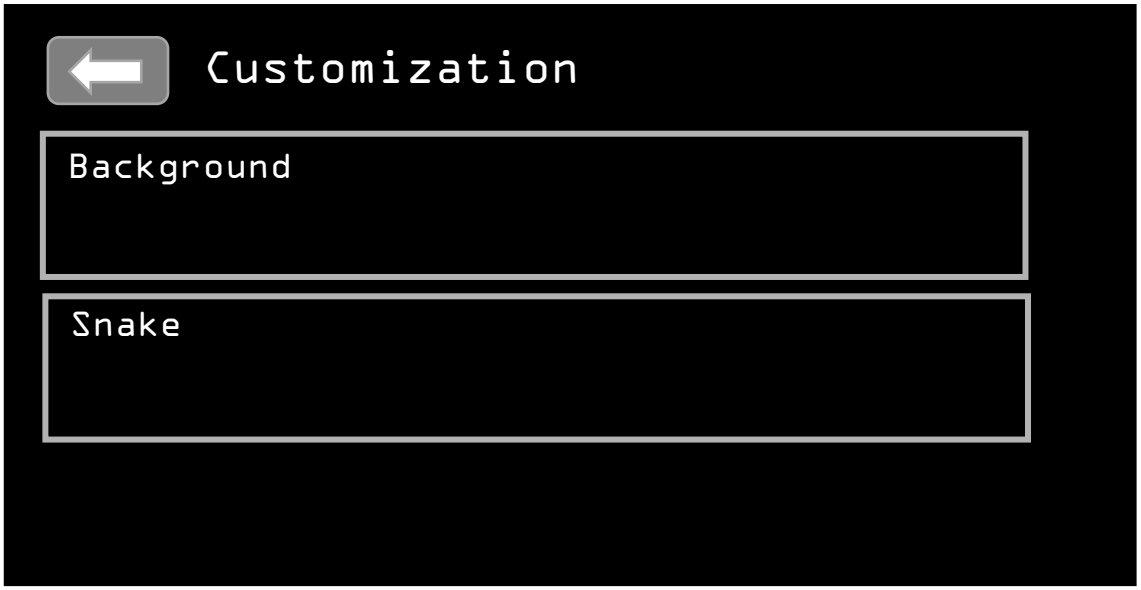


Figure 19: Design for Customization Menu

Progression Menu:

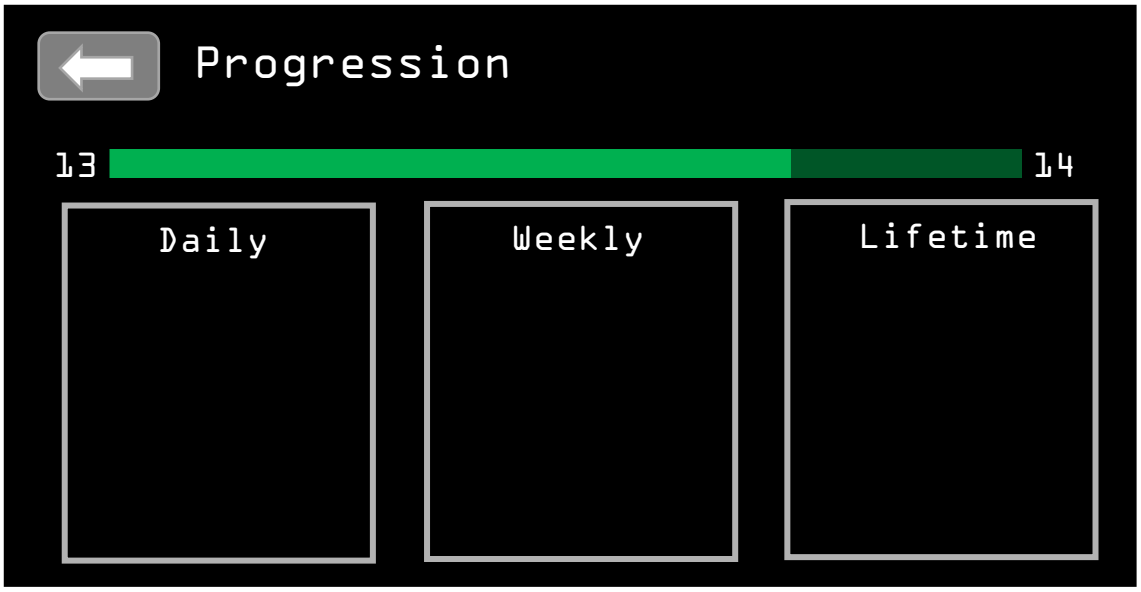


Figure 20: Design for Progression Menu

In-Match Interface:

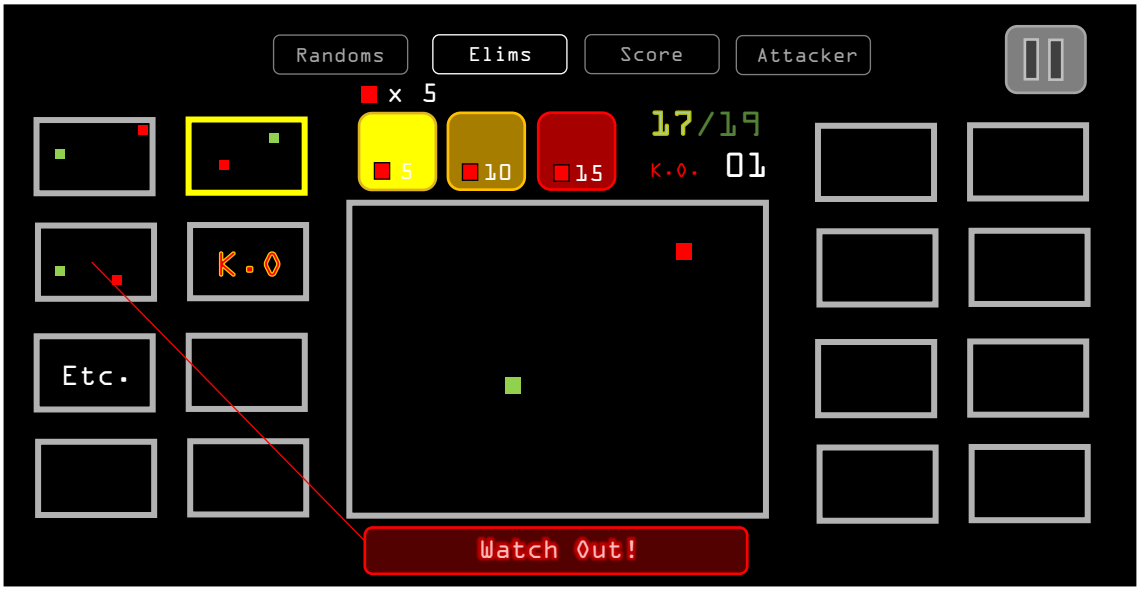


Figure 21: Design for In-Match UI

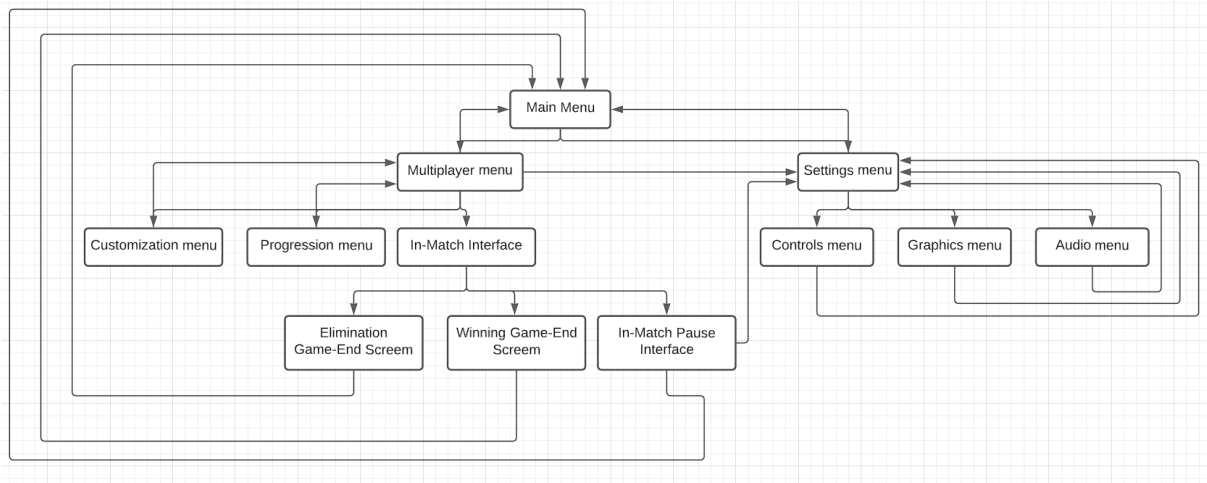
“Eliminated” Game End Screen:

Figure 22: Design for Elimination Screen

“Winner” Game End Screen:

Figure 23: Design for Winning Screen

Menu Flow System:



Hardware and Software Limitations

Hardware:

As stated in the analysis phase, a computer with a sufficient GPU to run either DirectX 9 or 11, a CPU that supports the SSE2 instruction set, sufficient storage and RAM will be capable of running this application. In addition, in order to access multiplayer features, computers with internet connections are required.

Software:

OS - An Operating System that is supported by the Unity Player is compulsory, as this will be what the game will run on, and so systems that cannot support the Unity Player cannot run the game. Fairly recent Windows, MacOS and Linux OSs are all compatible with Unity Player.

DirectX/OpenGL - Either DirectX or OpenGL needs to be installed on the system. These are Graphics APIs that assist in real-time rendering of 2D and 3D graphics by acting as a middleman to allow the game to seamlessly communicate with computer hardware. Both APIs are downloadable, but while DirectX is only available on Windows systems, OpenGL is open-source and is available on all major Operating Systems.

Development

In this section I will be developing the

Matchmaking

In this development phase, I wish to start working on the networking side of the game before I do the main game itself, as trying to implement networking on code that is specifically built for single-player is a very tough ask. This section will specifically be looking on getting users inside the matchmaking menus, getting them into rooms and starting games, as well as syncing all changes to all users in an active room.

Joining the Server

In Unity, scripts typically inherit from the *MonoBehaviour* class, which covers all general functionality of tasks in the engine. However, this does not include the networking functions that I will want to use.

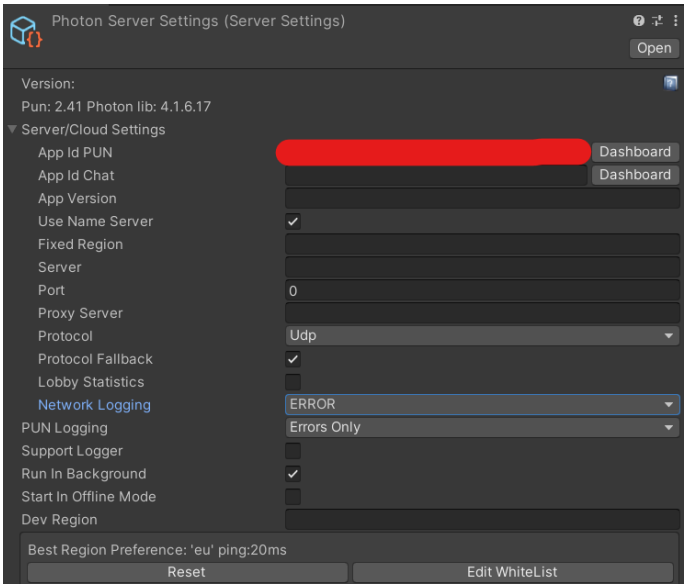
```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using Photon.Pun;
5   using TMPro;
6   using Photon.Realtime;
7
        Unity Script (1 asset reference) | 2 references
8   public class Launcher : MonoBehaviourPunCallbacks
```

To help overcome this issue, a third-party networking solution named Photon can be downloaded and used, which can attach onto Unity to provide extra functionality specifically for multiplayer purposes. In this case, adding on libraries that add Photon-specific functions, as well as extending the class from *MonoBehaviour* to *MonoBehaviourPunCallbacks* allow *callbacks* for creating rooms, joining rooms, error messages and more.

When a player first enters matchmaking, they will be sent to what is called the Photon Master Server.

```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    // Connects to the Photon Master Server as configured in the given Settings file
    Debug.Log("Connecting to Master");
    PhotonNetwork.ConnectUsingSettings();
}
```

This server allows the user to connect with other users to play with. Photon makes the process of customising how users connect to the master server quite simple with a menu the programmer can use within Unity's Inspector:



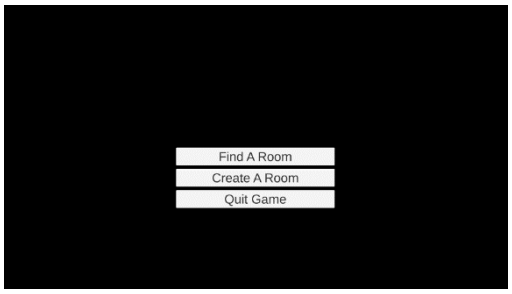
Upon successfully connecting to the master server, the user must still be able to join, create and leave rooms. This requires the user to join a lobby within this server, which is done using the *JoinLobby* function within the *OnConnectedToMaster* callback:

```
// Calls upon connecting to the Photon Master Server
3 references
public override void OnConnectedToMaster()
{
    Debug.Log("Connected to Master");
    PhotonNetwork.JoinLobby(); // Allows the User to create or join rooms
    PhotonNetwork.AutomaticallySyncScene = true;
}
```

Note that *AutomaticallySyncScene* is now set to true. This means that, when a player enters a room, all players in that room will be synced up. In short, this allows everyone in a room to transition to a game scene at the same time.

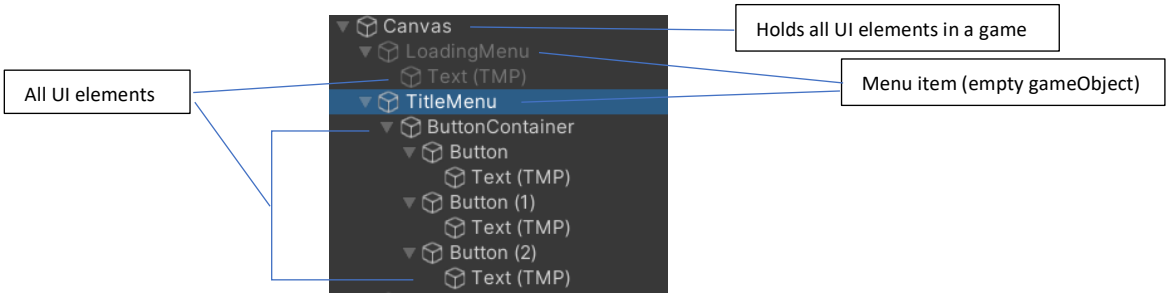
Loading and Lobby Menus

The process of joining a server and then a lobby needs to be represented to the user graphically somehow. To do this, a separate file can be made to control what menus are opened and closed which the *Matchmaking* file can call functions from. Before we do so however, the menus themselves have to be made, which can be done simply using Unity's UI configuration tools, which can be found within the inspector. The Loading and Lobby (right now called *TitleMenu*) menus look like this:



While the graphical representation of both these menus are quite lacklustre, these buttons and text act as placeholders to help make testing code easier as I develop, and I plan to refine and improve the design of the menus on further iterations.

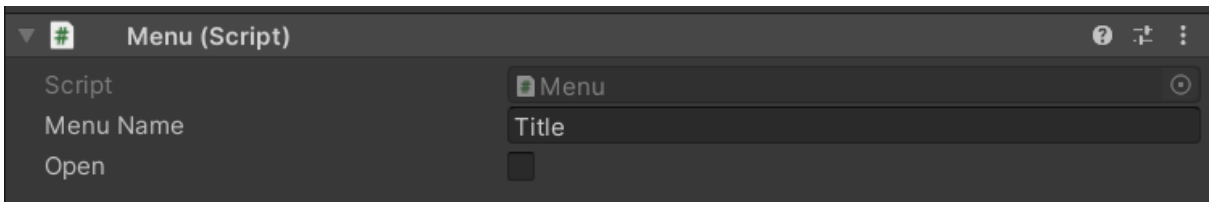
The control of the enabling and disabling of menus will be done using two scripts: *Menu* and *MenuManager*. The Menu script handles the switching of the physical menus upon certain triggers. When making each menu, all UI elements are made children of an empty gameObject, as follows:



To be able to switch menus, the program should be able to enable and disable the empty gameObjects within the canvas that hold their respective UI elements. This is not too hard to implement, as enabling/disabling gameObjects can be done with a built-in function from Unity:

```
Unity Script (6 asset references) | 3 references
public class Menu : MonoBehaviour
{
    public string menuName;
    public bool open;

    // Calls when a Menu needs to be opened
    2 references
    public void Open()
    {
        // Activate the Menu's parent gameObject
        gameObject.SetActive(true);
        open = true;
    }

    // Calls when a Menu needs to be Closed
    1 reference
    public void Close()
    {
        // Deactivate the Menu's parent gameObject
        gameObject.SetActive(false);
        open = false;
    }
}
```

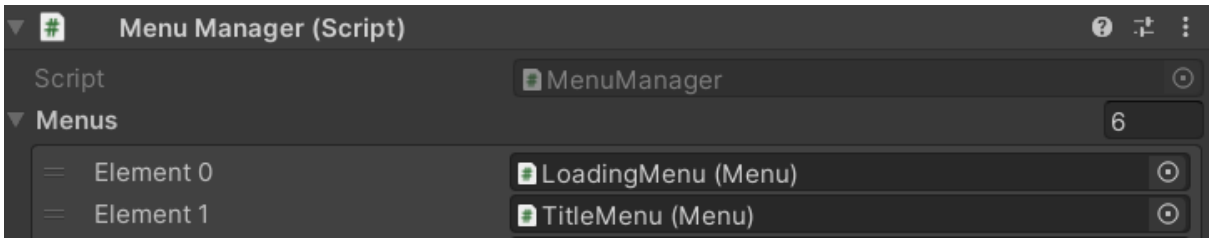
This script is placed in every parent menu gameObject, with the ability to name the Menu (under public variable *menuName*) in the Inspector:



The managing of which menus are opened upon certain triggers can be done with the script aptly titled *MenuManager*. As, unlike with *Menu*, there will only be one instance of this script in a scene, there has to be a way for this script to reference all menus in the scene. This is done by making a list of all gameObjects with the Menu class / holding the Menu script:

```
[SerializeField] Menu[] menus;
```

This list can be managed by adding the *MenuManager* script to the canvas holding all menu UI elements and adding all menu gameObjects to the list from the Inspector:

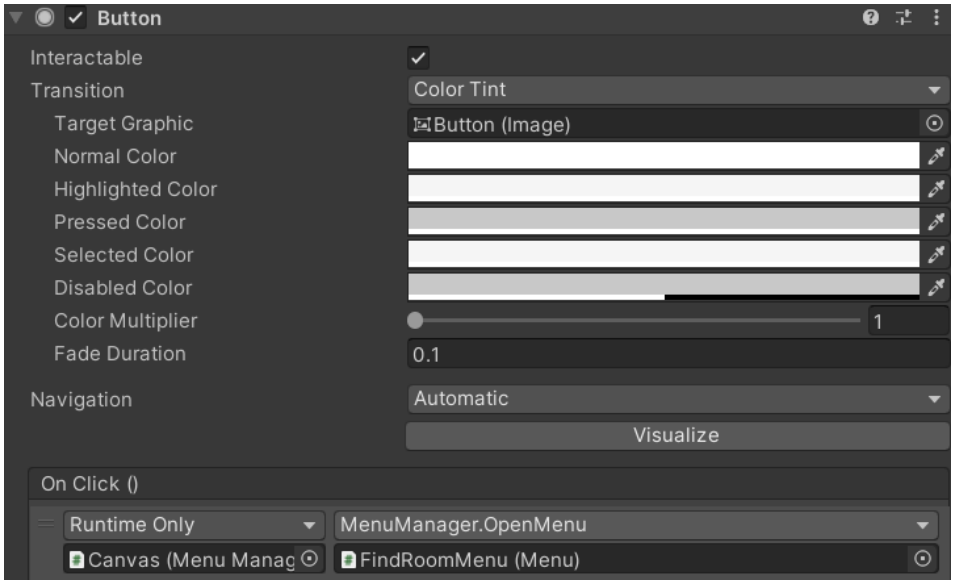


To open a certain menu, the program can go through every menu item in the *menus* list. If the menu being currently checked is the menu that needs to be opened, run the public *Open* function that that menu holds, which enables that menu's UI elements. If it isn't, and it is already open, run the public *Close* function the menu holds, which disables that menu's UI elements:

```
// Calls when a menu needs to be opened by string of menuName
7 references
public void OpenMenu(string menuName)
{
    for(int i = 0; i < menus.Length; i++)
    {
        if(menus[i].menuName == menuName)
        {
            // If the Menu incremented to is the menu we need, open it
            menus[i].Open();
        }
        else if (menus[i].open)
        {
            // If the Menu incremented to is not the menu we need and it is open, close it
            CloseMenu(menus[i]);
        }
    }
}

// Calls when a menu needs to be opened by an index in list menu
0 references
public void OpenMenu(Menu menu)
{
    // Works similar to the above function, just using the index of list menu rather than a string
    for (int i = 0; i < menus.Length; i++)
    {
        if (menus[i].open)
        {
            CloseMenu(menus[i]);
        }
    }
    menu.Open();
}
```

```
// Calls when a menu needs to be closed
2 references
public void CloseMenu(Menu menu)
{
    menu.Close();
}
```

One thing to point out is that there are in fact two *OpenMenu* functions, one taking in a string of the menu as an argument, and the other taking in the menu object itself as an argument. This is because in this game there can be two ways to open a menu. The function taking in the menu object as an argument can be used for times when an action is triggered (for example, pressing a button). Here, the inspector can customize which menu is opened from this trigger, as follows:



The function taking in a string as an argument is primarily used for calling the function in a script, where a physical trigger may not occur. For example, no user input may be provided to switch from a loading menu to a main menu. The function here has to check if the *menuName* variable of the menu its currently checking is the same as the provided argument string, and if so, enabling that menu UI.

To allow the functions in the *MenuManager* script to be called by any other script in the game. To do this, the script can be made into a Singleton. This can be done by setting a public static reference to itself, meaning that the variable it is referenced to is bound to the class of *MenuManager*, not the object in Unity:

```
public static MenuManager Instance;
```

```
Unity Message | 0 references
void Awake()
{
    // Allows this script to be called by other scripts
    Instance = this;
}
```

Having this class as a singleton means that it is both globally accessible and can only be instantiated once (which is good, as we only want one *MenuManager* per scene).

Going back to the Launcher Script, we can now override a Photon function called *OnJoinedLobby*, which calls when a player has successfully joined a lobby:

```
// Calls upon joining a Lobby
3 references
public override void OnJoinedLobby()
{
    // Opens the Title menu
    MenuManager.Instance.OpenMenu("Title");
    Debug.Log("Joined the Lobby");
```

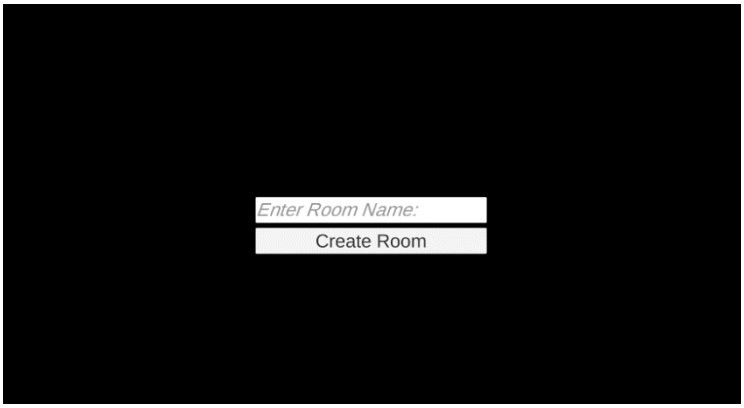
Now that *MenuManager* is a singleton, there is no need to reference the script in the *Launcher* script to access the functions needed to open menus, which is extremely useful, especially when more menus are made.

One last thing needed to do here is to allow the user to close the application upon clicking the "Quit Game" button, which can be done by simply calling Unity's *Application.Quit* function from a trigger:

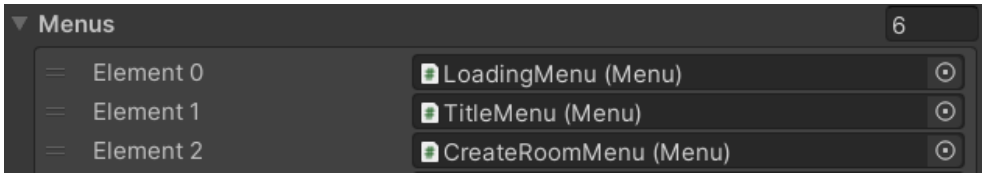
```
0 references
public void QuitGame()
{
    Application.Quit();
}
```

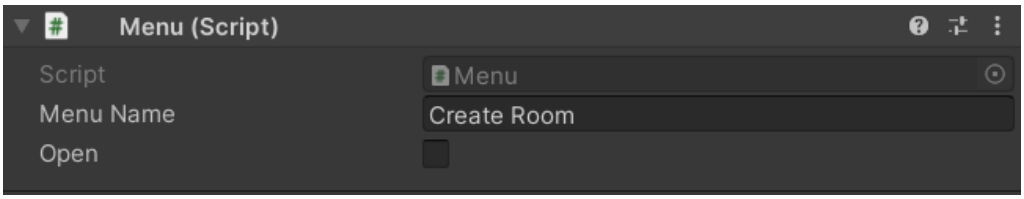
Creating a Room

Before code can be written to handle how players create rooms, the menu to create rooms itself must be made. For now, this will be a text box for user input as well as a button to confirm the user's choice to make a room.



The room creation menu gameObject will also refer to the Menu script and will be added to the array of menus for the *MenuManager* script:



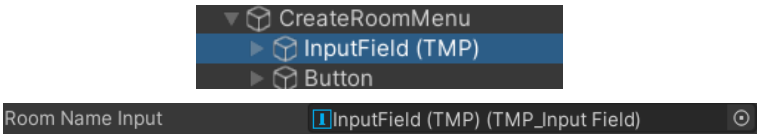


In addition, we have to add the ability to process information given from the input box. Unity does this with a library called TextMeshPro, which has to be imported using *Using TMPro;*. From here, a variable *roomNameInput* can be made to hold the value of the input box when the Create Room button is clicked, and Photon's *CreateRoom* function can be called, taking *roomNameInput* as an argument. A room will now be made in the server with the value of the input box as the room name.

```
[SerializeField] TMP_InputField roomNameInput;
```

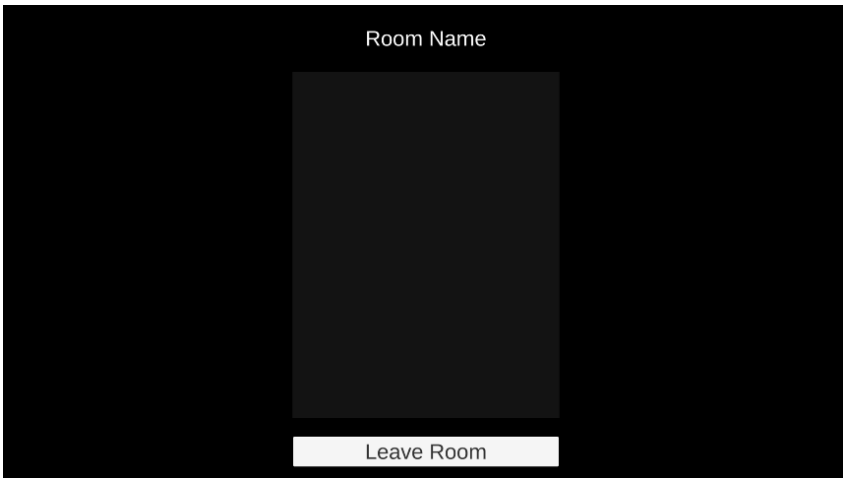
```
    PhotonNetwork.CreateRoom(roomNameInput.text);
```

In the Unity Inspector, the *roomNameInput* can be assigned to the respective Input Field by dragging the UI element from the hierarchy to the respective slot on the inspector:

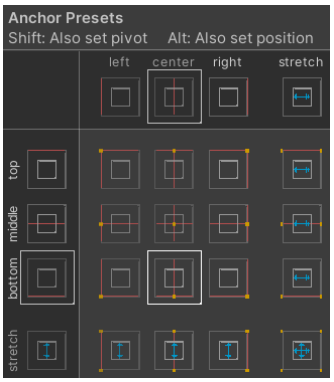


However, there is no visible indicator of a room being created to the user yet, which will be worked on next.

When a room is being created, one of two callbacks are made: either the room has been successfully created and the player has joined, or the creation has failed. In the former case, the user will be found in the room menu, looking as such:



Especially here, Unity helps with positioning UI elements by allowing us to anchor them to certain parts of the screen, automatically changing their size and scale according to programmer input.



This room will also be assigned the *Menu* script and be added to the list of menus in *MenuManager*.

In the *Launcher* Script, the function *OnJoinedRoom* that Photon provides (which is called when a player joins or creates a room) can be called here, and code can be added to open the room menu, as well as assign the name text box at the top of the screen to the room's name:

```
// Calls upon a player entering an existing room or a room they created
3 references
public override void OnJoinedRoom()
{
    MenuManager.Instance.OpenMenu("Room");
    roomNameText.text = PhotonNetwork.CurrentRoom.Name; // Show the room menu with the Title as the Room Name
```

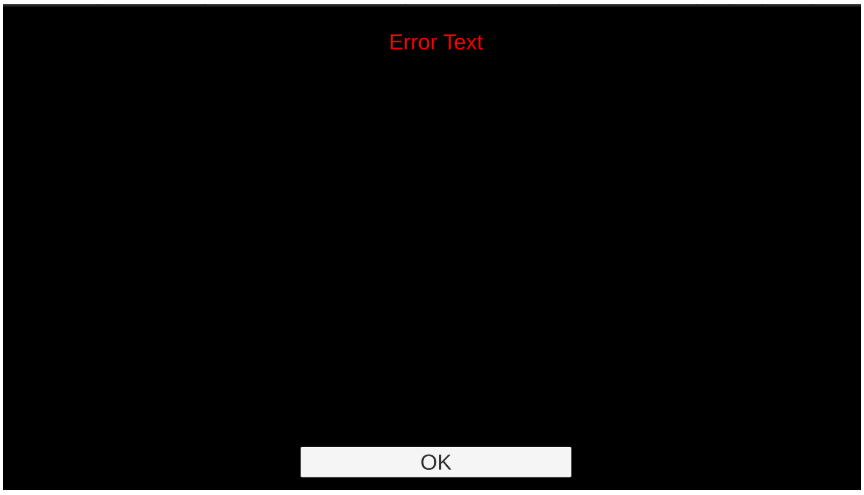
```
[SerializeField] TMP_Text roomNameText;
```

At this point, to leave the room, the user will have to tell the server that they are leaving the room and returning to the master server, and then will have to be transported back to the lobby menu. Notifying the server of the user leaving the room can be done by calling the *LeaveRoom* function, and then once *OnLeftRoom* is called notifying the completion of leaving the room, the Lobby menu can be opened:

```
// Calls when a player leaves a room (for the player only) (Button Trigger)
0 references
public void LeaveRoom()
{
    PhotonNetwork.LeaveRoom(); // Notify the server that the player has left
    MenuManager.Instance.OpenMenu("Loading"); // Make the player wait while they leave
}

// Calls when a player has finished leaving a room (local player only)
3 references
public override void OnLeftRoom()
{
    // Send the player to the Title Menu
    MenuManager.Instance.OpenMenu("Title");
}
```

If an error does occur when creating a room, the player must be redirected to a different menu telling them what error occurred, and directing them back to the lobby menu.

The lobby will be named, added to the menu list, and initially designed as such:



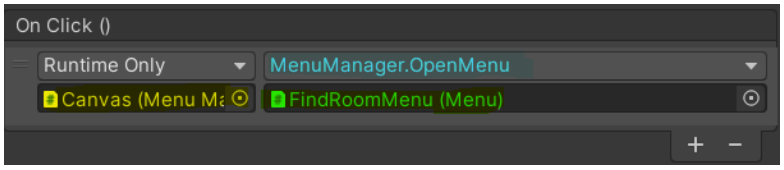
In the *Launcher* script, a function that Photon provides named *OnCreateRoomFailed* can be called. This function takes in both an error code and error message as arguments, which Photon provides automatically. Here, the function can send the user to the error menu and the error text UI element can be set to display the error message itself:

```
[SerializeField] TMP_Text errorText;
```

```
// Calls when there is an error in creating a room
3 references
public override void OnCreateRoomFailed(short returnCode, string message)
{
    errorText.text = "Room Creation Failed: " + message;
    MenuManager.Instance.OpenMenu("Error");
}
```

Assigning buttons

Up until this point, none of the buttons have been assigned yet to any functions. Fortunately, buttons can have functions assigned to them from the inspector, and, as most of the buttons thus far will be used for navigating between menus, this process can be done quickly and efficiently. For each button, the gameObject holding the script with the intended function within it can be dragged to the yellow highlighted section, the function itself can be selected from the blue highlighted dropdown, and if needed, another object can be assigned to the green highlighted area.

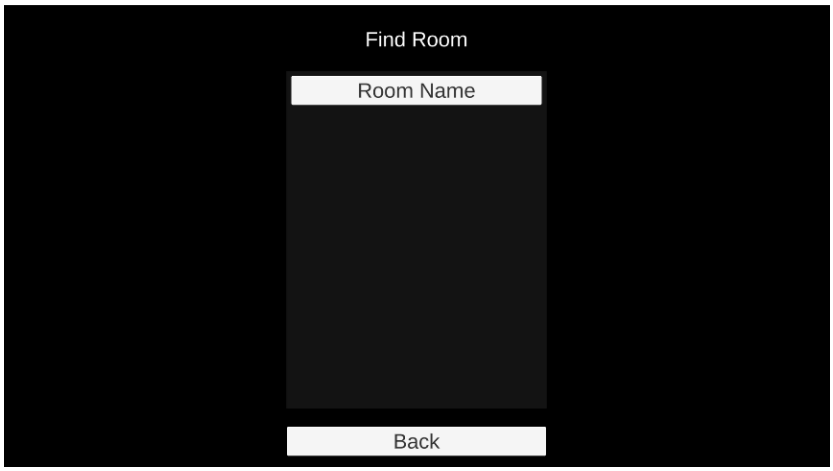


In this example, to have the first button in the lobby menu open the room finding menu, the canvas holding the *MenuManager* script can be dragged in, the *OpenMenu* function can be selected from the dropdown menu, and the *FindRoomMenu* gameObject can be dragged in to be confirmed.

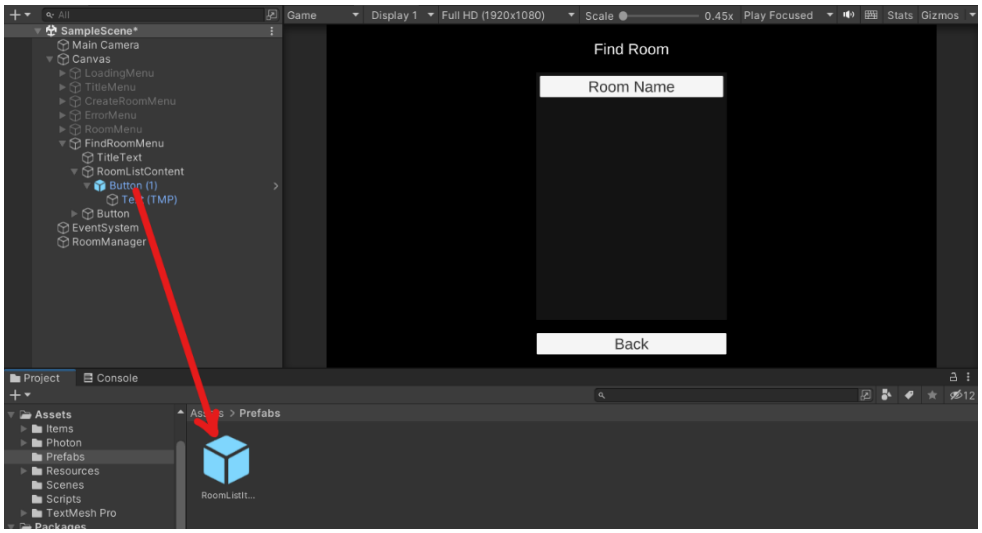
This can be done for all buttons in the scene so far, and for buttons I would implement in the future.

Finding a Room

Now that the ability to create a room has been completed, it is time to turn on to allowing players to find rooms. To do this, a menu that holds all rooms in the lobby has to be made. In the initial phases, this room can be made similar to the lobby menu, with one added item: a button indicating a room that has been created (the button named “Room Name”):



As this button is one that will be repeated depending on how many rooms there are in the lobby, it will have to be made into a prefab. Prefabs are configured gameObjects that can be saved within a project to be reused (or instantiated), and the button can be made into one by dragging it from the hierarchy to the project explorer, and renaming it (in this case, to *RoomListItem*):



Once that has been done, the prefab can have a script added to it. This script, also named *RoomListItem* controls what the button prefab does and what procedures it executes when it has been clicked.

To display the room’s name, the function *SetUp* takes in *_info* (the variable name for the simple information a room makes about itself when it is made, generally named *RoomInfo*), and sets the text element of the button to the name in the *_info* variable:

```
[SerializeField] TMP_Text text;

public RoomInfo info;

// Calls when a room list item has been instantiated
1 reference
public void SetUp(RoomInfo _info)
{
    // Sets the text UI to the room's name
    info = _info;
    text.text = _info.Name;
}
```

In the Launcher script, a function can be made to add or remove *RoomListItem* instances when a room is made or deleted. Photon has a callback for this called *OnRoomListUpdate* which can be added on to, and it takes a list of *RoomInfo* items for all the rooms active in the lobby. With this, we can run through the location of each *RoomListItem* instance in the menu, delete the instance at that location, and then instantiate new *RoomListItems* for every room still in the lobby, as well as set up the name for each of the new *RoomListItems*.

```
// Calls every time a room is created/destroyed (for all other players in the lobby)
5 references
public override void OnRoomListUpdate(List<RoomInfo> roomList)
{
    foreach(Transform trans in roomListContent)
    {
        // Destroy all room cards currently in the lobby
        Destroy(trans.gameObject);
    }
    for(int i = 0; i < roomList.Count; i++)
    {
        // add all room cards in the lobby list (if they have not been already removed)
        if (roomList[i].RemovedFromList) { continue; }
        Instantiate(roomListItemPrefab, roomListContent).GetComponent<RoomListItem>().SetUp(roomList[i]);
    }
}
```

This way, we make sure that if a player is in the finding room menu, but a room in that list provided to the player is deleted, the player will no longer see that room in the list of active rooms, reducing errors that may occur when joining a non-existent room.

In addition, we need to add a function that actually allows the player to join the room they select. This function, named *JoinRoom*, takes the *RoomInfo* of the room the player selected and makes the player join the room with the same name as the info provides.

```
// Calls when a player joins a room (local player only)
1 reference
public void JoinRoom(RoomInfo info)
{
    // Notify the server that the player has entered the room and make the player wait until they have fully joined
    PhotonNetwork.JoinRoom(info.Name);
```

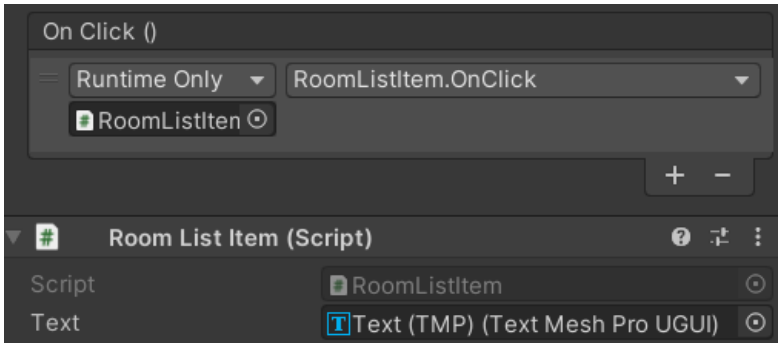
However, to allow this function to be called on the click of the *RoomListItem* button, we need to make the *Launcher* script a singleton, much like *MenuManager*:

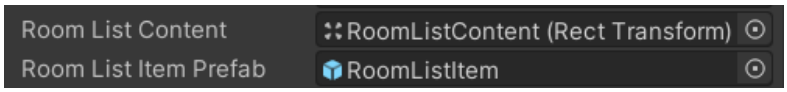
Now, back in the *RoomListItem* script, we can call the *Launcher* script's *JoinRoom* function, taking in the room's info variable as the argument:

```
// Calls when the button is clicked
0 references
public void OnClick()
{
    // Calls the JoinRoom function in the Launcher Script
    Launcher.Instance.JoinRoom(info);
}
```

Now, all that is needed to do is assign the *OnClick* function to the *RoomListItem* prefab from the Unity Inspector, as well as assign functions to any buttons that will carry them out that have not had functions assigned to them yet (refer to the Assigning Buttons section). We will also need to tell the *RoomListItem* script where the text it will manipulate is, and tell the *Launcher* Script where the *PlayerListItem* prefab is, as well as where these prefabs will be placed in the scene (indicated by the transform variable of the room list):

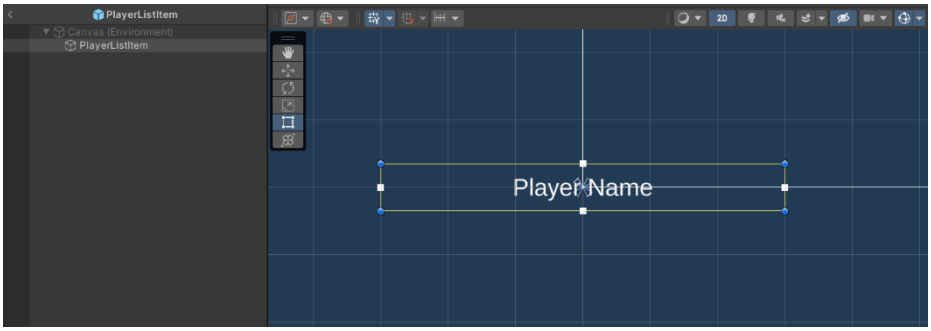
```
public static Launcher Instance;

Unity Message | 0 references
void Awake()
{
    // Allows this script to be called by other scripts
    Instance = this;
}
```





Displaying players in a Room

Displaying any players in an active room can be done in a similar way to displaying rooms in a lobby. A prefab here has to be made to show player information, but instead of it being a clickable button, the information can be displayed in a simple text element:



A script named *PlayerListItem* will be responsible for the display itself of each *PlayerListItem* prefab. In this script, displaying the name of each player is very similar to displaying the name of each room in the *FindRoomMenu*, this time taking in player information as the argument to the function:

```
Unity Script (1 asset reference) | 2 references
public class PlayerListItem : MonoBehaviourPunCallbacks
{
    [SerializeField] TMP_Text text;
    Player player;

    // Calls when a player list item has been instantiated
    2 references
    public void SetUp(Player _player)
    {
        // Sets the text UI to the player's name
        player = _player;
        text.text = _player.NickName;
    }
```

Notice *_player.Nickname*. As, so far, the name of a player has not actually been set, to act as a placeholder, I would like to generate a random 4-digit number to identify a player. To do this, the *OnJoinedLobby* function in the *Launcher* script can be expanded to add a procedure that assigns each player with a number:

```
// Calls upon joining a Lobby
3 references
public override void OnJoinedLobby()
{
    // Opens the Title menu
    MenuManager.Instance.OpenMenu("Title");
    Debug.Log("Joined the Lobby");

    // Assign each player a random number
    PhotonNetwork.NickName = "Player " + Random.Range(0, 1000).ToString("0000");
}
```

The *PlayerListItem* script will also be capable of removing a *PlayerListItem* prefab if that player leaves the room. To do this, the *gameObject* holding the *PlayerListItem* prefab can be “destroyed” when the Photon function *OnPlayerLeftRoom* is called. Here, as we only want the player who left’s prefab to be destroyed, every prefab will check if the player that left was the player that prefab represented, and if so, it will destroy itself:

```
// Calls when a player has left a room (takes the player that left as an argument)
3 references
public override void OnPlayerLeftRoom(Player otherPlayer)
{
    // If the player that left
    if(player == otherPlayer)
    {
        Destroy(gameObject);
    }
}
```

Back in the *Launcher* script, the *PlayerListItem* prefab and the position each prefab is in will need to be referenced in the script to be put in the inspector later:

```
[SerializeField] Transform playerListContent;
[SerializeField] GameObject playerListItemPrefab;
```

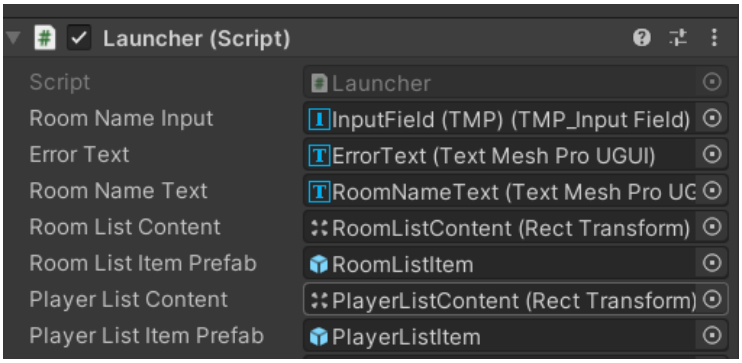
Next, a new Photon function needs to be expanded upon: *OnPlayerEnteredRoom*, taking the player as an argument. Here, like when making a new room button in the *FindRoomMenu*, the *PlayerListItem* prefab will need to be instantiated and have its *SetUp* function initialised:

```
// Calls when a Player enters a room
3 references
public override void OnPlayerEnteredRoom(Player newPlayer)
{
    // Create a new player card and add it to the room
    Instantiate(playerListItemPrefab, playerListContent).GetComponent<PlayerListItem>().SetUp(newPlayer);
}
```

When a player first enters a room they joined, they must also be able to see all players in that room. To do this, every player in the room must have their *PlayerListItem* prefab instantiated for the player that joined (to get information on every player that is in the room, a list can be made taking in Photon's pre-made list of players). This must be done when a player joins the room, and thus will be placed within the *OnJoinedRoom* function:

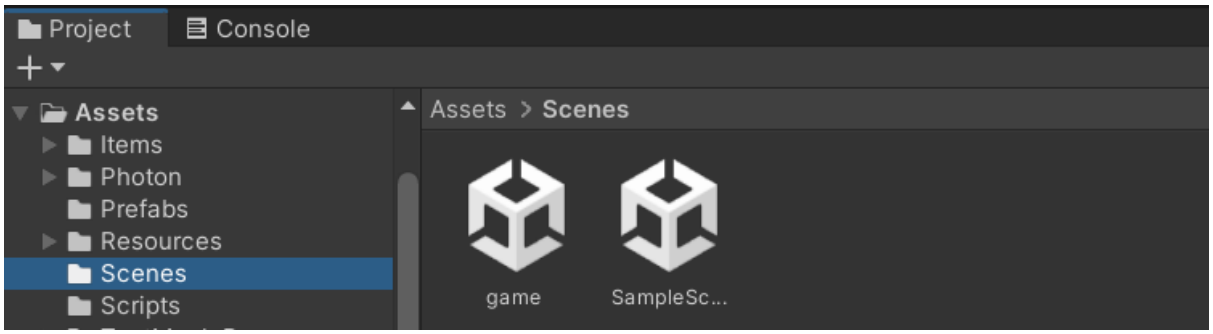
```
// Calls upon a player entering an existing room or a room they created
3 references
public override void OnJoinedRoom()
{
    MenuManager.Instance.OpenMenu("Room");
    roomNameText.text = PhotonNetwork.CurrentRoom.Name; // Show the room menu with the Title as the Room Name

    // Make a new list of all player in the room when joining
    Player[] players = PhotonNetwork.PlayerList;

    for (int i = 0; i < players.Length; i++)
    {
        // Instantiate a card for each player in the lobby
        Instantiate(playerListItemPrefab, playerListContent).GetComponent<PlayerListItem>().SetUp(players[i]);
    }
```

In the Unity inspector, the *PlayerListItem* prefab as well as where it will be placed are assigned to the *Launcher* script, and the prefab itself will refer to the *PlayerListItem* script:

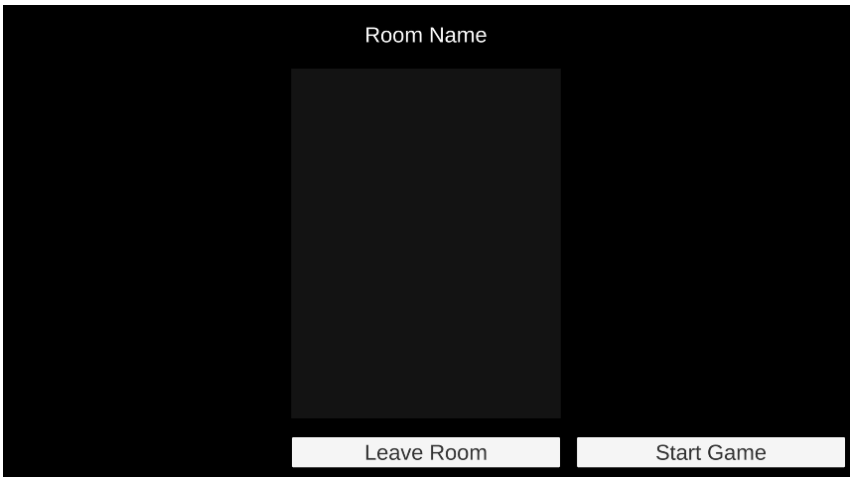


Starting the Game

The final part of this iteration is to ensure that a host of the room is able to send themselves as well as all players in a room from the room to the game. First, a new scene (named *game*) needs to be made to hold the game world itself:



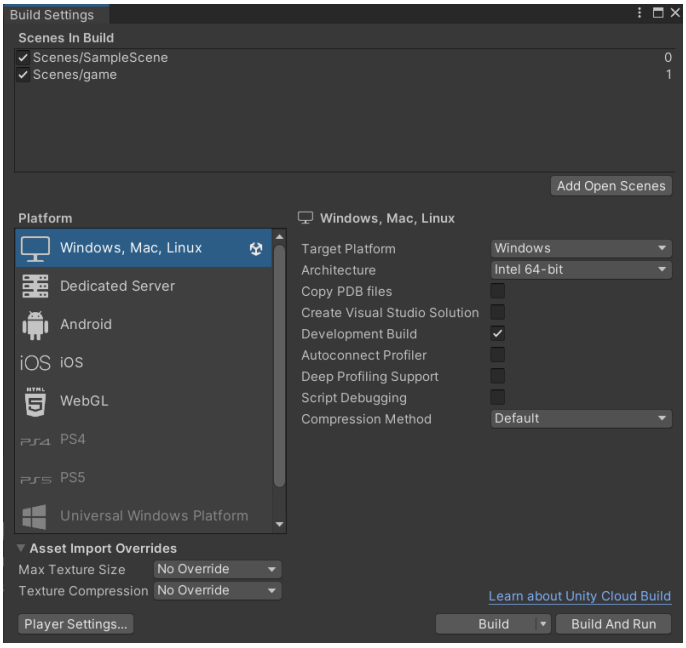
Then, in the *RoomMenu*, another button can be added to start the game:



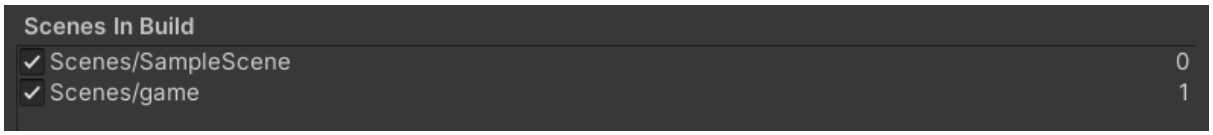
In the *Launcher* script, a function *StartGame* can be made to be called when the "Start Game" button is clicked. This function loads the scene at index 1:

```
// Calls upon starting the game
0 references
public void StartGame()
{
    // Load the game Scene
    PhotonNetwork.LoadLevel(1);
}
```

Understanding the indexing of scenes in Unity requires the Build menu to be open in Unity, which can be done by pressing Ctrl+Shift+B. This opens the following menu:



To run multiple instances of a game (as well as to produce a distributable version of the finished game), the project needs to be built. As of right now, all that matters is what is at the top:



Here, the scenes that are in the project can either be enabled or disabled (be made inaccessible) in the build by clicking the checkbox on the left of the scene name. On the right, however, is the index of the scene. This scene index is accessible by script, and is what *PhotonNetwork.LoadLevel* takes as an argument.

As seen before in the *OnConnectedToMaster* function, *AutomaticallySyncScene* is set to true when a player enters a lobby. This ensures that when a *LoadLevel* is triggered, all players switch from the menu scene to the game scene at the same time. However, for now, we only want the ability for one person to make that decision to start the game.

To do this, the *Launcher* script must first refer to the "Start Game" button in the room menu. Then, in the *OnJoinedRoom* function (which calls whenever a player joins/creates a room), the "Start Game" button can be set to be active, only if the player in the room is the Master Client/the host:

```
[SerializeField] GameObject startGameButton;
```

```
startGameButton.SetActive(PhotonNetwork.IsMasterClient); // Allow the Room Host to start the game when needed
```

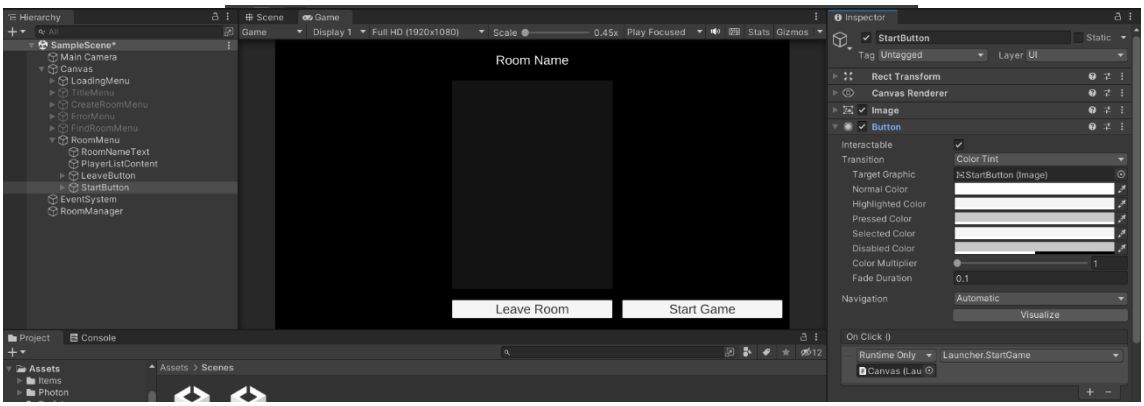
Now, if there is a player in a room, the host will be able to start the game.

Host Migration

However, an issue that may come into play is if the host decides to leave the lobby. Thankfully, in this case, Photon has in-built host migration, which means that if the host leaves, a random player in the lobby will be the host. Upon the host switching, the function *OnMasterClientSwitched* is called in the *Launcher* script, which we can use to set the "Start Game" button to be active for the new host:

```
// Calls when the active host leaves the lobby
3 references
public override void OnMasterClientSwitched(Player newMasterClient)
{
    // Set a random player as the new host
    startGameButton.SetActive(PhotonNetwork.IsMasterClient);
}
```

Now that host migration has been taken care of, in the Unity inspector, the *Launcher* script can refer to the Start Button gameObject, and the button itself can call the *StartGame* function in the *Launcher* script:



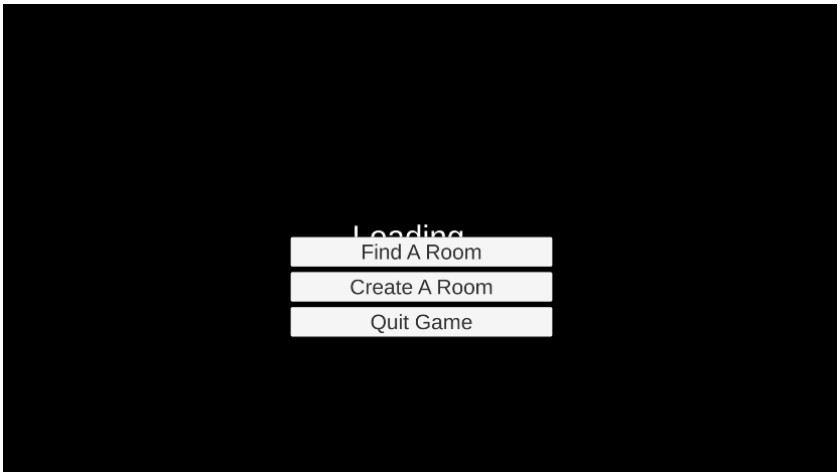
Testing current project

Testing the code that I have made so far will most be done within the inspector, as well as built versions of the project. The main function of this testing is that every input the user puts in gives the expected output.

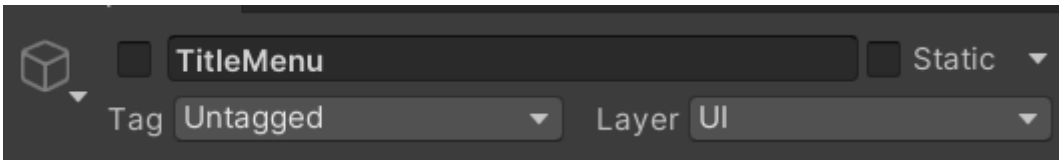
When starting the game, the following tests are done:

Test	Expected Output	Actual Output
Starting the game	Loading menu opens, followed by the Title menu opening and the Loading menu closing	*Loading menu and Title menu opened at the same time*

The first issue I came across when testing this out was that, upon starting the game, both the Loading and Lobby menus opened at the same time:



Upon checking the inspector, I found that the Loading menu and the Lobby menus are enabled by default. I had to disable the Lobby menu gameObject within the inspector before starting the game to ensure that both menus are not open at the same time.



Testing this process now produced this result:

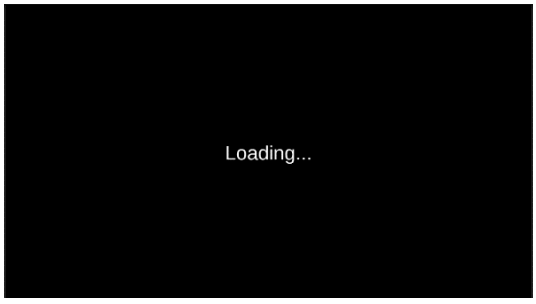
Test	Expected Output	Actual Output
Starting the game	Loading menu opens, followed by the Title menu opening and the Loading menu closing	*The loading menu does not close when needed, so both menus are open at once*

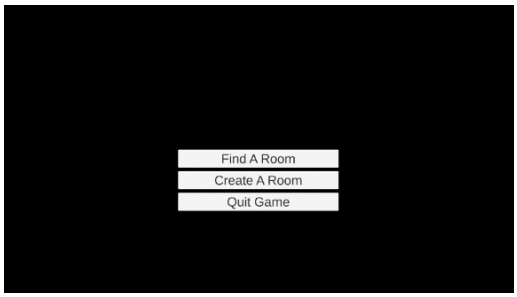
What I found was that, even with that precaution, once the user joined the lobby, they still would find themselves with both loading and lobby menus at the same time, like beforehand.

Upon checking the inspector, I realized that once a player enters a lobby, the *MenuManager* script is trying to close the loading menu. However, as all menus have their respective *Open* booleans set to false by default, there is no room indicated to the script to disable, causing the lobby menu to open without closing the loading menu, having both appear at once.

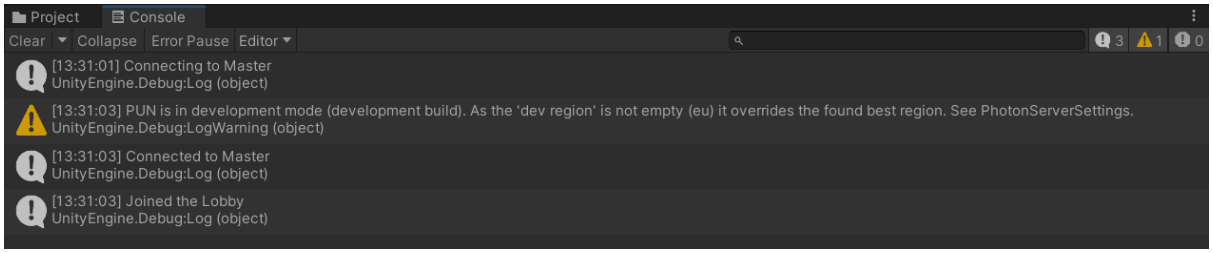
Fixing this issue was a matter of going to the Inspector and setting the *Open* boolean to true when the game starts (which can be done as public variables are able to be viewed and edited in the Inspector). Now, the *MenuManager* can find that the loading menu is open and promptly close it.

Now, when testing the code written, I could see that the two menus opened and closed as expected:



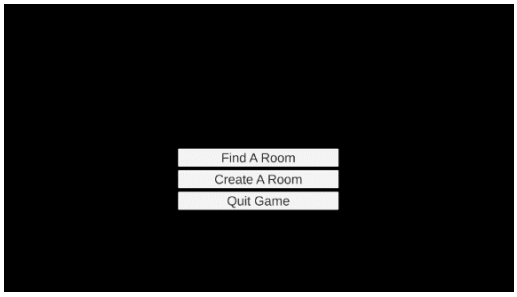


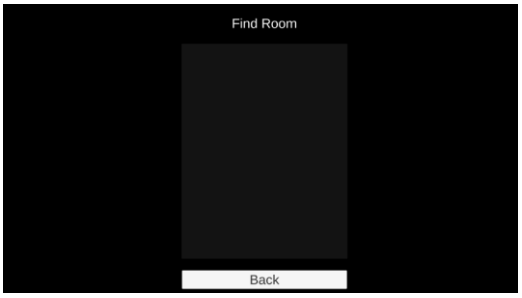
In addition, I could confirm that the code worked as intended, as the debug messages I added into functions to indicate when the master server/lobby has been joined were outputting correctly:



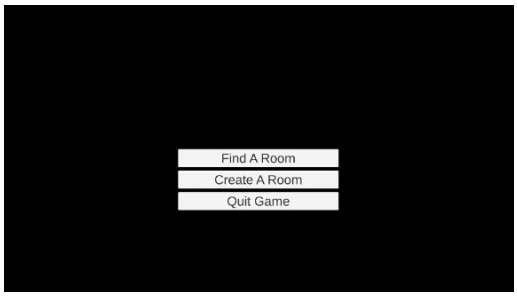
Further testing went as follows:

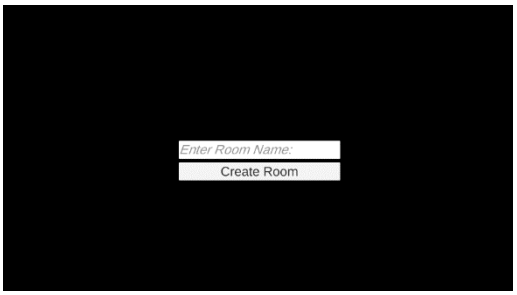
Test	Expected Output	Actual Output
Pressing the “Find a Room” Button	Player is sent to the find room menu	*PASS*
Pressing the “Back” button in the Find Room Menu	Player is sent back to the Title Menu	*PASS*





Test	Expected Output	Actual Output
Pressing the “Create a Room Button”	Player is sent to the create room menu	*PASS*





Test	Expected Output	Actual Output
Pressing the “Quit Game”	Game exits	*Nothing Occurs*

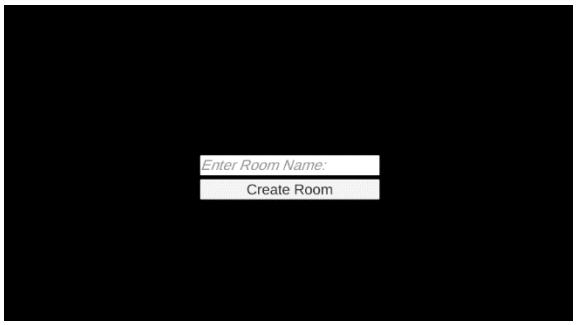
Here, the quit game function does not appear to actually exit the game within the inspector. Upon viewing documentation, I realised that, within the Unity Editor, exiting the application can only be done via the play button at the top of the editor. By building the game, I can see that the quit button indeed works as intended.

Testing functions in relation to creating rooms went as such:

Test	Expected Output	Actual Output
Pressing the “Create Room” button with an empty input box	Nothing happens	*Room created with random room name*

When initially creating a room, one case I did not come across is the situation where the user leaves the room name input field empty. The room will still be created, but there will be a very long replacement name to the room. This may not impose any errors that stop the program just yet but

looking for rooms where some of the names are not understandable may be quite troublesome for the user.

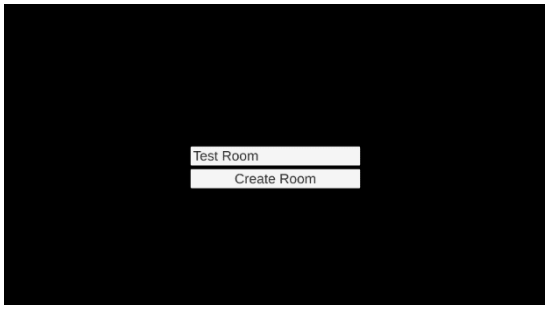


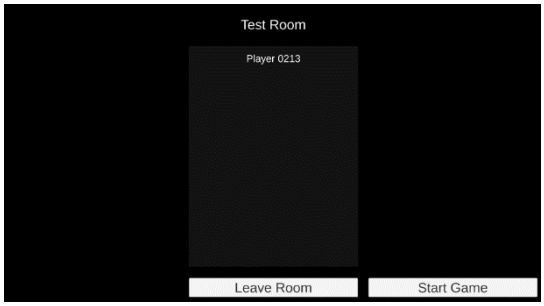
To solve this, I can add a check to see if the input provided is empty and, if so, returning nothing. This means that, if the given input is null, the player will not experience any changes when creating a room, ensuring they always put in a valid room name.

```
// Calls upon a player creating a room (button press trigger)
0 references
public void CreateRoom()
{
    if (string.IsNullOrEmpty(roomNameInput.text))
    {
        return; // Do not run this function if the room name is empty
    }
    PhotonNetwork.CreateRoom(roomNameInput.text);
```

Now, testing the code:

Test	Expected Output	Actual Output
Pressing the “Create Room” button with an empty input box	Nothing happens	*PASS*
Pressing the “Create Room” button with a named input box	Room Menu opened and room with inputted name created	*PASS*





Further testing went as follows:

Test	Expected Output	Actual Output
Pressing the “Leave Room” button	Player sent back to Title Menu	*PASS*
Pressing the “Start Game” button	All players sent to Game Scene	*PASS*

Additional code changes

These changes to code were made after the testing phase after further thought was put into how these functions work, upon seeing unpredicted processes in my code.

First off was the fact that if a player was to leave a room and then create another one (on their own), there would seem to be two instantiations of the *PlayerListItem* prefab, instead of only one. Upon checking the code, this was because when the player joins a new room, existing *PlayerListItem* prefabs from the last room carried over, and were not deleted.

To fix this, I add code to the *OnJoinedRoom* callback, to allow for all existing *PlayerListItem* prefabs to be destroyed before new ones are added:

```
// Calls upon a player entering an existing room or a room they created
3 references
public override void OnJoinedRoom()
{
    MenuManager.Instance.OpenMenu("Room");
    roomNameText.text = PhotonNetwork.CurrentRoom.Name; // Show the room menu with the Title as the Room Name

    foreach (Transform child in playerListContent)
    {
        Destroy(child.gameObject); // Remove existing player names from menu
    }
```

Another consideration that was made was checking if a room was full or not. For now, I plan on only letting a maximum of 4 players in a room at any one time (although that is subject to change) for development purposes. However, when joining a room, there is no checks being done to ensure that the number of players in a room does not exceed 4.

To fix this, I first add a check to the OnPlayerEnteredRoom callback. If, when the player joins, there is now 4 players in the room and the player receiving this callback is the Master Client, then that player can force the room to be invisible and inaccessible to players outside the room:

```
// Calls when a Player enters a room
3 references
public override void OnPlayerEnteredRoom(Player newPlayer)
{
    // Create a new player card and add it to the room
    Instantiate(playerListItemPrefab, playerListContent).GetComponent<PlayerListItem>().SetUp(newPlayer);

    if(PhotonNetwork.CurrentRoom.PlayerCount == 4 && PhotonNetwork.IsMasterClient)
    {
        // If there are 4 players in a room, make the Master Client close the room to those outside the room
        PhotonNetwork.CurrentRoom.IsVisible = false;
        PhotonNetwork.CurrentRoom.IsOpen = false;
    }
```

We also need to allow players to join when the number of players in a room is less than 4 upon a player leaving the room. To do this, I add a new Photon callback OnPlayerLeftRoom, and check if there is now less than 4 players in the room and the player receiving this callback is the Master Client. If so, that player can force the room to be visible and accessible to players outside the room.

```
3 references
public override void OnPlayerLeftRoom(Player otherPlayer)
{
    if (PhotonNetwork.CurrentRoom.PlayerCount < 4 && PhotonNetwork.IsMasterClient)
    {
        // If there are less than 4 players in a room, make the Master Client open the room to those outside the room
        PhotonNetwork.CurrentRoom.IsVisible = true;
        PhotonNetwork.CurrentRoom.IsOpen = true;
    }
}
```

Now, testing the code, when entering a new room, players do not see duplicates of their name, and a maximum of 4 players are now allowed per room.

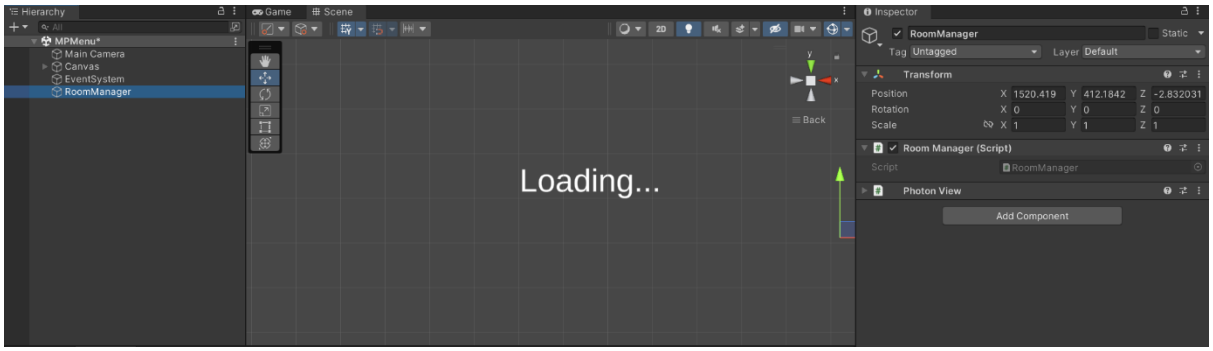
Spawning

This section will look into getting players spawned into the game itself so that they are able to start moving.

Instantiating prefabs

To handle persistent player data, spawning as well as receiving data from other players, a *PlayerManager* prefab will be instantiated upon the game starting, while at the same time, a *PlayerController* prefab will be instantiated that will handle the movement and actions of each player themselves. This architecture is needed in order to allow players to communicate with each other in some way. To allow for these two to be instantiated, we will have a *RoomManager* prefab that will be instantiated which instantiates a *PlayerManager* when a player joins a room, and that *PlayerManager* will instantiate the *PlayerController*, as well as handle destroying and instantiating this prefab from here on out. When a player leaves the *PlayerManager* and *PlayerController* will automatically be destroyed by Photon and only *RoomManager* will remain.

To start off, I create an empty GameObject in my Matchmaking scene and call it *RoomManager*. I will also add a script to it also named *RoomManager*:



To allow the script to handle Instantiating *PlayerManager* prefabs across a network, I will make sure to import the *Photon.Pun* library to the script and have it inherit from *MonoBehaviourPunCallbacks*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Photon.Pun;
using UnityEngine.SceneManagement;
using System.IO;

Unity Script (1 asset reference) | 1 reference
public class RoomManager : MonoBehaviourPunCallbacks // Extending the class from MonoBehaviour to allow photon Callbacks
```

All that is needed to be done here is to check if the local player has switched scenes and, if so, instantiate the *PlayerManager*. First, I will make this *RoomManager* class a singleton, by checking if another *RoomManager* already exists (and if so, destroying itself), and if not, setting itself the instance and allowing itself to stay active even when loading new scenes using *DontDestroyOnLoad*:

```
public static RoomManager Instance;

Unity Message | 0 references
private void Awake()
{
    if (Instance) // If another RoomManager exists..
    {
        Destroy(gameObject);
        return; // Destroy itsef
    }
    DontDestroyOnLoad(gameObject); // Ensure there is only one RoomManager
    Instance = this;
}
```

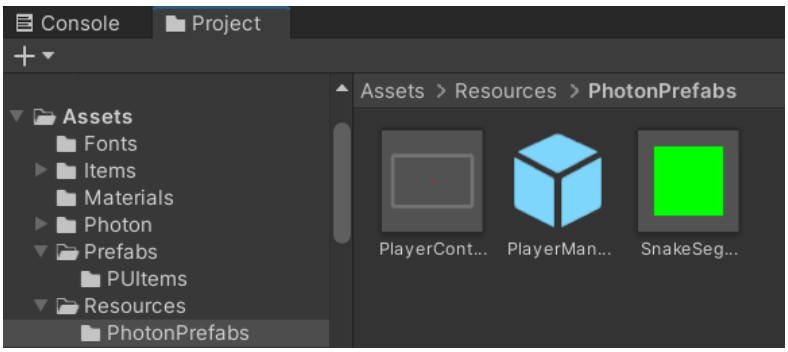
Then, overrides for *OnEnable* and *OnDisable* callbacks will be implemented. In either one, the base methods of these functions need to be called as they are essential for Photon to function correctly (note that these two functions are the only callbacks that need a base method called to work). After that, importing the *UnityEngine.SceneManagement* library to the script, the *OnEnable* and *OnDisable* will subscribe and unsubscribe from the callback *sceneLoaded* with a new method *OnSceneLoaded*, which takes in arguments Scene and *loadSceneMode*. Subscribing and unsubscribing essentially means that whenever a scene is switched, the *OnSceneLoaded* function can be called:

```
Unity Message | 3 references
public override void OnEnable() // Calls when the RoomManager is enabled
{
    base.OnEnable(); // Base OnEnable method is called
    SceneManager.sceneLoaded += OnSceneLoaded; // Subscribe to the sceneLoaded callback with the OnSceneLoaded method
}

Unity Message | 4 references
public override void OnDisable() // Calls when the RoomManager is disabled
{
    base.OnDisable(); // Base OnDisable method is called
    SceneManager.sceneLoaded -= OnSceneLoaded; // Unsubscribe to the sceneLoaded callback with the OnSceneLoaded method
}
```

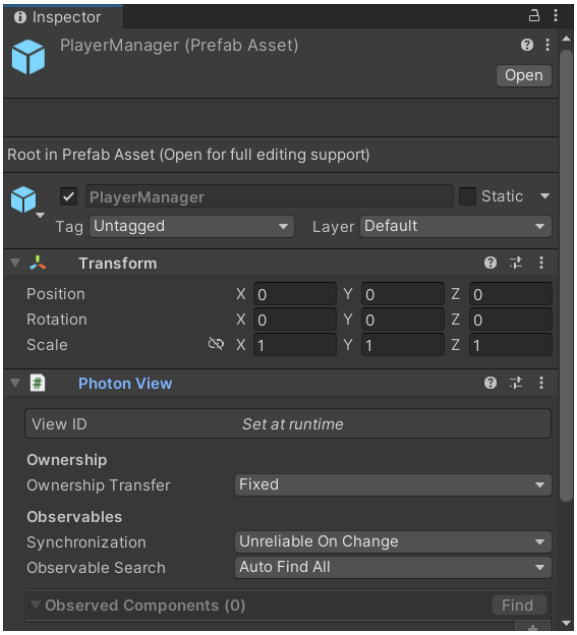
Within the *OnSceneLoaded* function, we can check if the scene the player is switching to is the game scene by checking if the index of the scene the player switches to is the same index as the game scene. If so, we instantiate the *PlayerManager* prefab using Photon. Doing this is different from instantiating normally using Unity.

Since the prefab will need to be spawned on other people's computers, we reference it via a string which indicates the name of the prefab, rather than a reference to the prefab object in the editor like normal instantiation. All prefabs being instantiated with Photon need to be the resources folder of my project. Since these prefabs are not referenced in the editor in the final build, rather with a string, we need them to be included in this build even with this drawback. In unity, this is done by putting these prefabs in the resources folder, which will always be included in the final build whether the prefabs themselves have been referenced or not.

I put Photon-specific prefabs in their own folder named PhotonPrefabs just for better organisation:



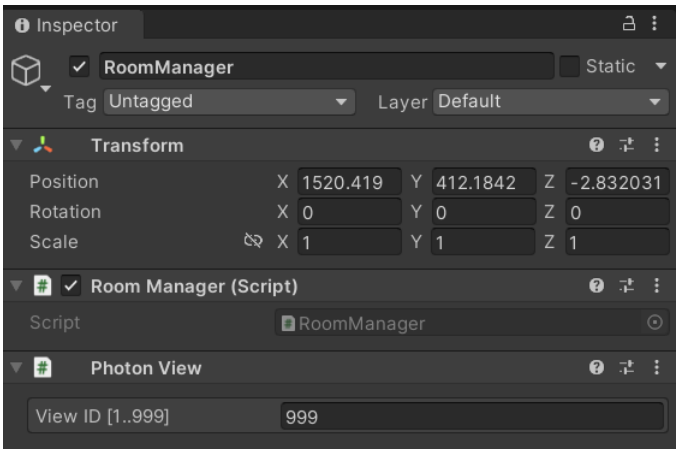
I then will make a new prefab here called *PlayerManager*, to which I will add a Photon View component. This component allows these prefabs to sync with other people's games over a network. The *PlayerManager* is made into a prefab the same way *RoomListItem* was, and then a Photon View component is also added on to the *RoomManager* GameObject, to allow it to instantiate Photon prefabs:



Back in the *RoomManager* script, we can now write code to instantiate the *PlayerManager* prefab, ensuring to specify where exactly the prefab is in our assets folder using *path.combine* (which combines 2 strings to get a file path), and allowing it to spawn in the centre of the scene (the prefab could be spawned anywhere due to it being visually empty, but having it in the centre is easier to implement):

```
2 references
void OnSceneLoaded(Scene scene, LoadSceneMode loadSceneMode) // Calls when a player's scene is changed
{
    if(scene.buildIndex == 2)
    {
        // If the player switched to the game scene, instantiate the PlayerManager prefab
        PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs", "PlayerManager"), Vector3.zero, Quaternion.identity);
    }
}
```

In the Editor, we set the *RoomManager's* Photon ViewID to 999, as, when players enter a lobby, the ViewID (unique identifier) they get is incremented, so setting the *RoomManagers* ID to the max value ensures that there will be minimal collisions in identifiers:



In the *PlayerManager* prefab I will add a script also named *PlayerManager* to it. Inside the script, I will import the *Photon.Pun* library and make a new *PhotonView* variable, assigning it to the Player's *PhotonView* when the script is initialised (within the Awake function):

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Photon.Pun;
using Photon.Realtime;
using System.IO;

Unity Script (1 asset reference) | 0 references
public class PlayerManager : MonoBehaviour
{
    PhotonView PV;

    Unity Message | 0 references
    void Awake()
    {
        PV = GetComponent<PhotonView>();
    }
```

After this, the script will check what index the local player is in the list of players in a room by iterating through a list of all players in the room and checking if the player in the list is the local player:

```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    // initialise a list of players, an indicator of the local player, as well as an indicator of the index of the local player
    Player[] RoomPlayers = PhotonNetwork.PlayerList;
    Player LocalPlayer = PhotonNetwork.LocalPlayer;
    int PlayerIndex = 0;
    // Check what index in the list of players in a Room the local player is in
    for (int i = 0; i < RoomPlayers.Length; i++)
    {
        if (RoomPlayers[i] == LocalPlayer)
        {
            PlayerIndex = i;
        }
    }
```

Note how this snippet of code is put within the Start function, not Awake. This is due to two things:

- In its first call, the Awake function typically runs before the Start function does, so the player would need to find out what PhotonView it has before it can do anything in relation to communication with other players or the server
- Awake can be called multiple times (every time the script is initialised, whether or not it is active), while Start can only be called once in the script (only when it is first initialised), and so the index of the player only needs to be checked once, as it will not have to be used for long.

Also within the Start function, the script does a check to see if the player running the script is the local player, and if so, calls a function named *CreatePlayer* that instantiates the *PlayerController* at a certain position in the Game scene:

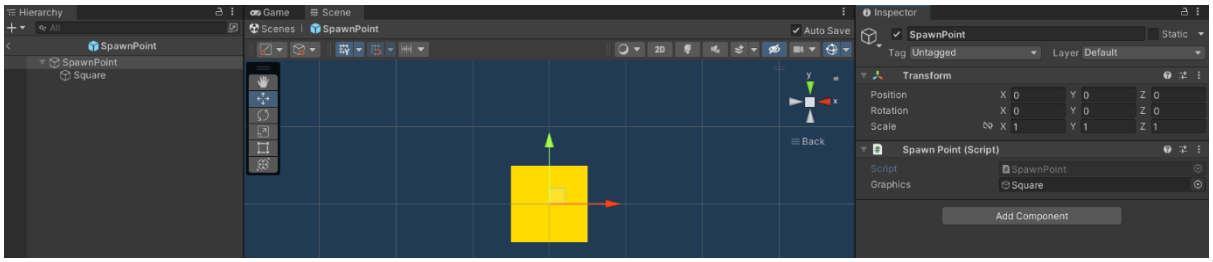
```
    if (PV.IsMine) // If this is the local player:
    {
        // Get the transform of the PlayerCintroller's spawn via a separate script
        Transform SpawnPoint = SpawnManager.Instance.GetSpawnPoint(PlayerIndex);
        CreatePlayer(SpawnPoint);
    }
}

1 reference
void CreatePlayer(Transform s)
{
    // Spawns the PlayerController at the transform position and rotation provided (Quaternion.identity = no rotation)
    PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs", "PlayerControllerMain"), s.position, Quaternion.identity);
}
```

Finding Spawnpoints

The finding of a player's spawnpoint is done on a script named *SpawnManager*, held on a GameObject in the Game Scene of the same name. This GameObject has a Photon View of its own, to ensure it is able to communicate with other players. As there only should be one of these in each game, the script is a singleton:

```
public class SpawnManager : MonoBehaviour
{
    public int PlayerIndex;

    public static SpawnManager Instance;

    public SpawnPoint[] SpawnPoints;

    public Player[] RoomPlayers;

    PhotonView PV;

    Unity Message | 0 references
    void Awake()
    {
        PV = GetComponent<PhotonView>();
        RoomPlayers = PhotonNetwork.PlayerList; // Grabs a list of all players in the room
        Instance = this; // Makes this script a singleton
    }
```

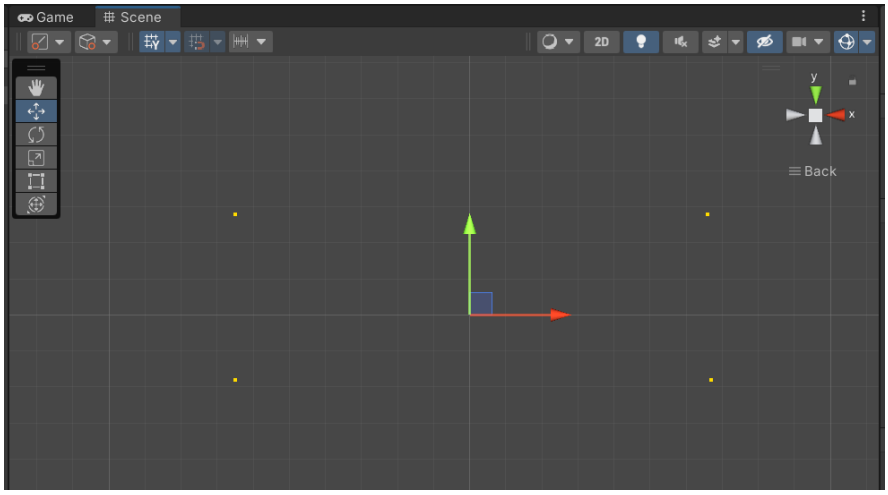
Before figuring out where a player is to be spawned, a prefab must be made indicating a room's spawnpoint. As this prefab will only indicate the places a player can spawn in, it does not need to be synced across players and thus can be instantiated without Photon. The visual for the spawnpoint indicator is just a simple yellow square as such:



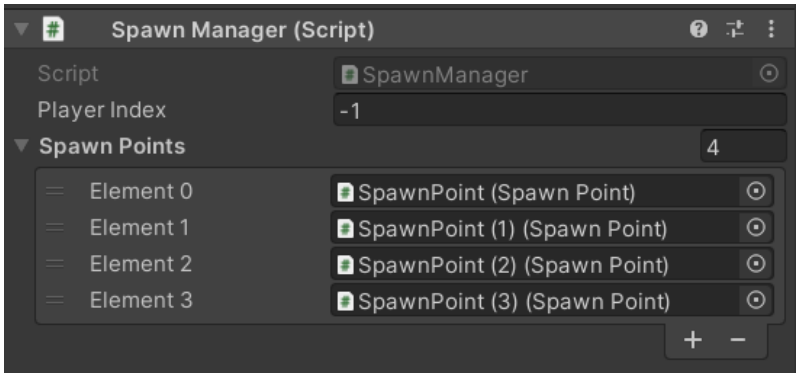
It can be seen that this prefab has a script attached to it. All this script does is disable the graphics for the prefab when it is initialised, so that the position of it isn't lost, but the players do not see the spawnpoint visual itself:

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
    Unity Script (1 asset reference) | 1 reference
5   public class SpawnPoint : MonoBehaviour
6   {
7       [SerializeField] GameObject graphics;
8
        Unity Message | 0 references
9       void Awake()
10      {
11          graphics.SetActive(false); // Set the graphics renderer of the Spawnpoint indicator to false
12      }
13  }
```

Once the prefab for a spawnpoint has been made, they can be added to the game scene like so:



These spawnpoints can then be added to the list of spawnpoints in the *SpawnManager*:

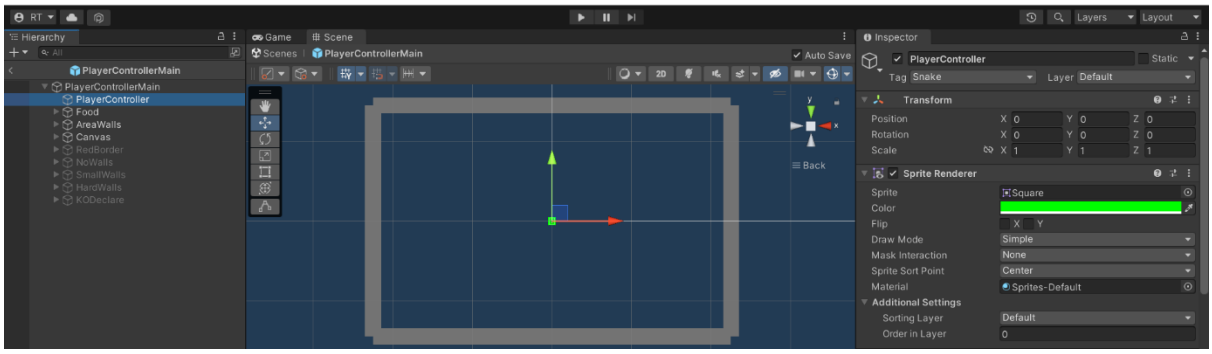


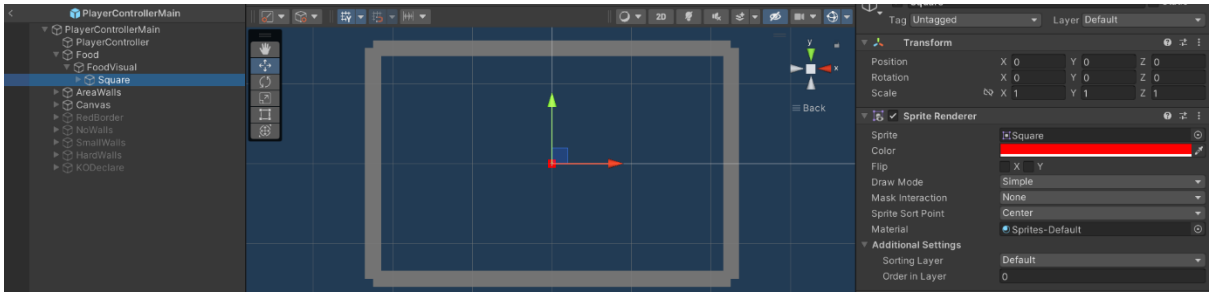
Once this has been done, the *GetSpawnPoint* function that was called by the *PlayerManager* can be coded in. This function takes in the index of the player in the list of players in the room and returns the transform of the spawnpoint in the list of spawnpoints with the same index:

```
1 reference
public Transform GetSpawnPoint(int i)
{
    // Output (in the debug console) and return the transform of the corresponding spawnpoint to the player calling the function
    Debug.Log(SpawnPoints[i].transform.position + " By player " + RoomPlayers[i]);
    return SpawnPoints[i].transform;
}
}
```

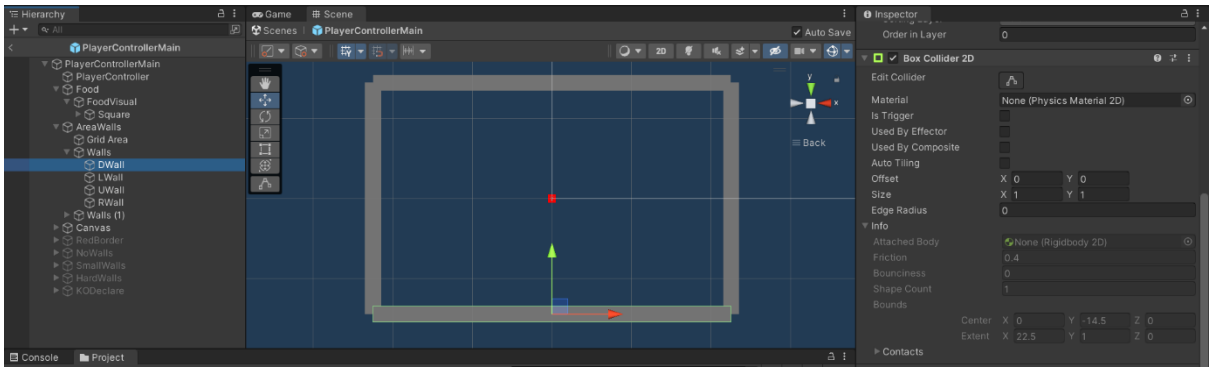
Creating the PlayerController Prefab

Now that spawning for the *PlayerController* is coded, a prefab for the *PlayerController* itself needs to be made. First, a sprite for the Player's snake, as well as food, are made (ensuring that they have box colliders in them, meaning that they trigger a collision function whenever another box collider interacts with it):

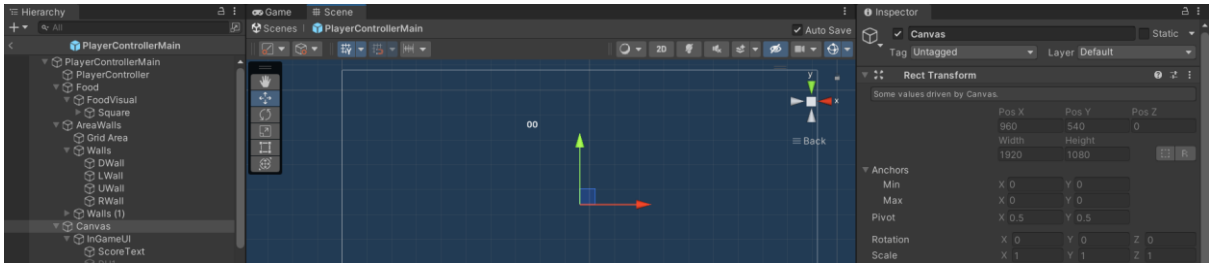




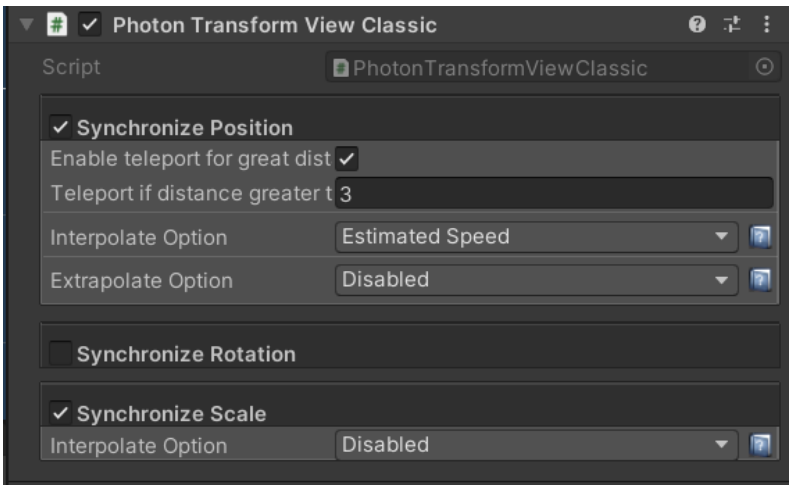
Then, the walls containing the player are made, also containing box colliders themselves:



Finally, a canvas showing basic UI elements such as the number of food a snake will collect will be made:



All GameObjects in this prefab that need to be synced across all players in the room will then have a Photon View added to them. In addition, they will have a Photon Transform View added to them, which allows them to sync any changes in position, rotation and scale to all other players in the room.



Note that the Interpolate Option for Position here is “Estimated Speed”. This means that, while the GameObject attached to it moves, this script estimates how fast it is moving, and mimics that movement and speed to all other players in the room. In the case of the Snake and the Food, as they

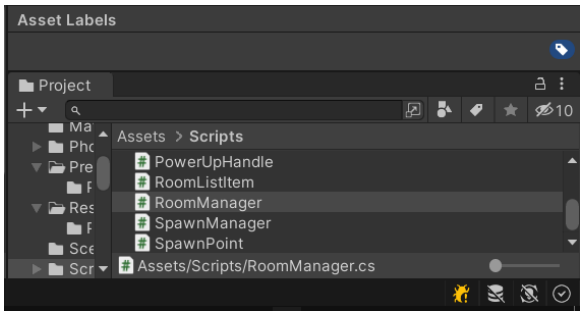
will not move smoothly in game, the Interpolation option for Position, Scale and Rotation is set to “Disabled”.

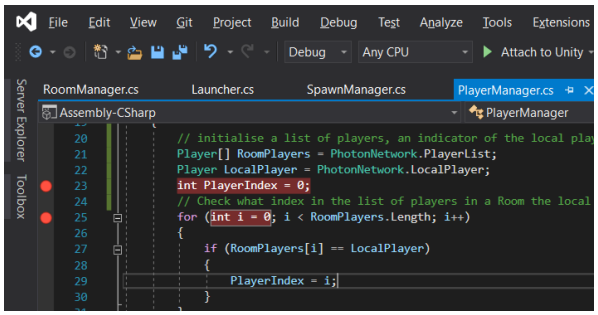
Testing current project

There are not many tests done on this part of the project, as most of this process is automated. However, the few tests done are as follows:

Test	Output
PlayerManager Instantiated	*PASS*
GetSpawnPoint Called	*PASS*
PlayerController Instantiated	*PASS*

Testing that these functions work is done by adding breakpoints to certain lines indicating the calling of these functions and checking if they are triggered. This is done by clicking the bug icon on the bottom right of the Unity editor to toggle the debugger, and then going to the Visual Studio application and clicking the “*Attach to Unity*” button.





This builds the code and, should it build successfully, you should be able to use breakpoints by clicking on the left of the corresponding line number, highlighting the code line red and showing a red circle where you clicked. When the code passes that highlighted line, it will stop the code at that point, allowing the programmer to check variables at that point, and then continue with the code. In this case, breakpoints allow me to confirm that the functions I am testing for do indeed get called.

Movement, Food, Death and Winning

In this section I will code in mainly the *PlayerController* and *Food* scripts, adding functionality for the player to move, as well as collecting Food, as well as handling collisions which would cause the player to lose and other players to win.

Player Movement

To keep track of all the segments of a player’s snake at any one time, we set up a list of Transforms (positions in the game scene). When a player starts the game (within the Start function) , the player’s Transform will be added to this list:

```
// Make a new list of Transforms and add the position of the player to it
_segments = new List<Transform>();
_segments.Add(this.transform);
```

An indicator of the direction of the snake also needs to be recorded to allow the player to change the direction. This will be done in both a vector2 (a vector variable in 2 dimensions x and y) as well as a string. The vector variable will be set to a unit vector to the right, while the string variable will be set to “None”. Both of these will also be done when the script starts:

```
// Reset the direction of the player
_direction = Vector2.right;
DirectionIndicator = "None";
```

In terms of actual movement, a coroutine is created. Coroutines differ from typical functions in the sense that they can wait for a certain amount of time before continuing with their processes. As the player’s snake will not be moving constantly and rather over increments of time, there needs to be a way for the movement function to wait a set period of time before moving the snake again. This is done using yield return new *WaitForSeconds*() (shown below), which passes a float indicating the time – in seconds – to wait for.

```
0 references
IEnumerator Movement()
{
    while (PV.IsMine) // Runs as long as the player running this function is the local player
    {
        // Set each snake segment's position as the position of the segment in front of it
        for (int i = _segments.Count - 1; i > 0; i--)
        {
            _segments[i].position = _segments[i - 1].position;
        }
        // Set the front of the snake's position as one unit in the direction of the variable set
        this.transform.position = new Vector3(
            Mathf.Round(this.transform.position.x) + _direction.x,
            Mathf.Round(this.transform.position.y) + _direction.y,
            0.0f
        );
        // Wait a set period of time before repeating this process
        yield return new WaitForSeconds(repeat_time);
    }
```

The actual movement of the snake itself consists of, first, moving all segments that may already exist on the snake to the position of the segment in front of it, and then moving the head of the snake to the position one unit in the direction of the _direction vector variable. The program then waits for *repeat_time* seconds before repeating the function again.

The changing of directions of a player’s snake is also done within a coroutine. Here, the script checks if a user has pressed any of the directional inputs provided. If so, the script sets the new direction for both the string and vector variables based on which of the 4 directional inputs the user pressed, then waits a given amount of time before allowing the user to change direction again. If the user has not pressed a button at any time, the code will yield return 0, which essentially means the code will wait until the next frame before moving to the next line (or in this case, running the function again). We, again, want to make sure that the direction of the snake can only be done by the local player tied to that snake, so we check if the Photon View of the player running the code is that of the local player:

```
0 references
IEnumerator Direction()
{
    // Only run this for the local player
    while (PV.IsMine)
    {
        // Change direction of the snake according to player input, only if the direction inputted is not the opposite of the current direction
        if (Input.GetKeyDown(KeyCode.W) && DirectionIndicator != "Down")
        {
            _direction = Vector2.up;
            DirectionIndicator = "Up";
            yield return new WaitForSeconds(repeat_time / 1.5f);
        }
        else if (Input.GetKeyDown(KeyCode.S) && DirectionIndicator != "Up")
        {
            _direction = Vector2.down;
            DirectionIndicator = "Down";
            yield return new WaitForSeconds(repeat_time / 1.5f);
        }
        else if (Input.GetKeyDown(KeyCode.A) && DirectionIndicator != "Right")
        {
            _direction = Vector2.left;
            DirectionIndicator = "Left";
            yield return new WaitForSeconds(repeat_time / 1.5f);
        }
        else if (Input.GetKeyDown(KeyCode.D) && DirectionIndicator != "Left")
        {
            _direction = Vector2.right;
            DirectionIndicator = "Right";
            yield return new WaitForSeconds(repeat_time / 1.5f);
        }
        // If nothing is inputted, wait until the next frame and try again
        yield return 0;
    }
}
```

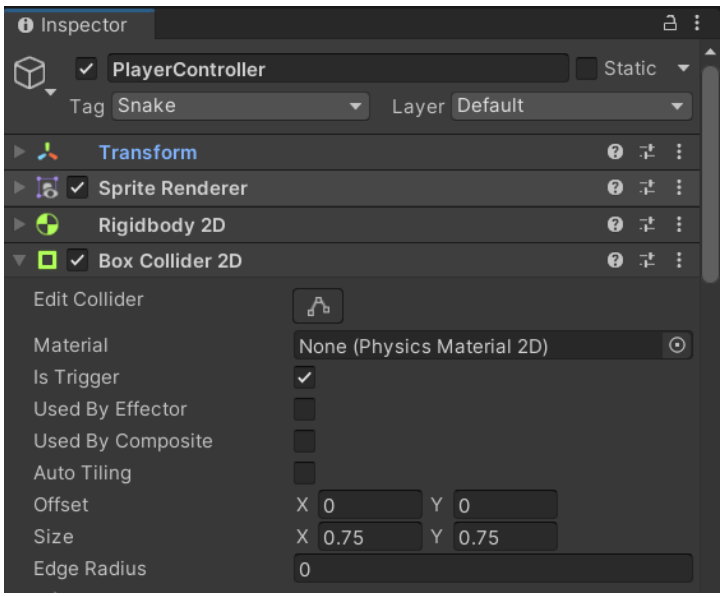
Notice that we add an extra check whenever a player does press down a directional key, to see if the current direction is not the opposite of the inputted direction. This is to ensure the player does not move backwards on itself and die immediately.

Unlike typical functions, that need to be called again and again to run multiple times, coroutines can be started and stopped, and as long as there is a while statement that allows the coroutine to loop, starting a coroutine can allow it to run continuously until it is told to stop. We can start both the direction and movement coroutines within the Start function:

```
// Start the named coroutines
StartCoroutine("Movement");
StartCoroutine("Direction");
```

Food

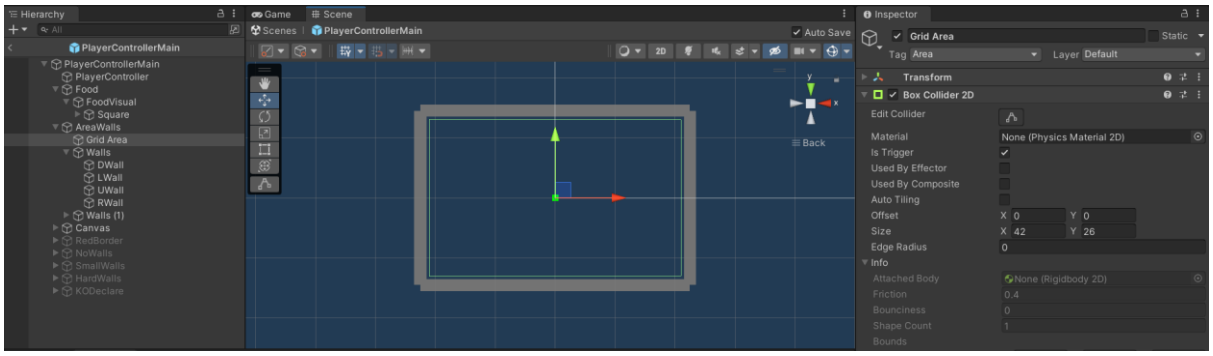
As part of the game, players are required to consume food they run through to gain score. We can detect whenever a certain GameObject collides with another GameObject as long as both of them



have box colliders, and at least one of these colliders is a trigger (meaning it calls a function when collided with). In this case, I set the *PlayerController* snake head as a trigger collider as this will be the object running into either food or walls.

The food of a player automatically is active when a *PlayerController* prefab is spawned, but it still must be able to move around randomly when being touched by the player's snake head. To do this, I assign a new script *FoodScript* to the food GameObject to control all of its actions.

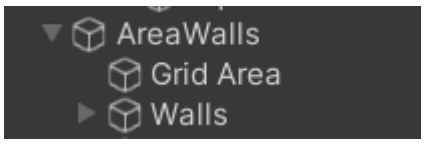
Before coding anything, I will need to create the area that the food will randomly move around. This will be done by creating an empty GameObject named Grid Area as a child of the *AreaWalls*. This GameObject will only have a Box Collider component in it (shown in green below) to define the area the food will use:



First off in this script, I will set a GameObject variable called *AreaWallsObj* that will hold the *AreaWalls* of the player, as well as a BoxCollider2D variable named *gridArea*. This will be found by automatically finding the Box Collider component of the *GridArea* GameObject in the Awake function of the Food code using *GetComponentsInChildren*. This function looks through every child of the GameObject it refers to, checking for the first component of the type passed through it:

```
Unity Message | 0 references
void Awake()
{
    // Grab the first BoxCollider2D component within the child objects of the Area Walls gameobject
    gridArea = AreaWallsObj.GetComponentInChildren<BoxCollider2D>();
    // Move the food gameobject to a random position
    RandomPos();
}
```

In this case, each child GameObject in *AreaWallsObj* is looked through:



As the Grid Area GameObject is the first child in the Hierarchy of *AreaWalls*, that is looked through first. This GameObject holds a BoxCollider2D component, and so *gridArea* will automatically hold this component as its value.

After doing so, the next function, named *RandomPos*, is called in the Awake function. This function gets the bounds (the minimum and maximum x and y values) of the *gridArea* Box Collider, and gets random values of x and y values within these bounds. These x and y values then become the new coordinates of the food object, rounded to ensure only whole number values are used:

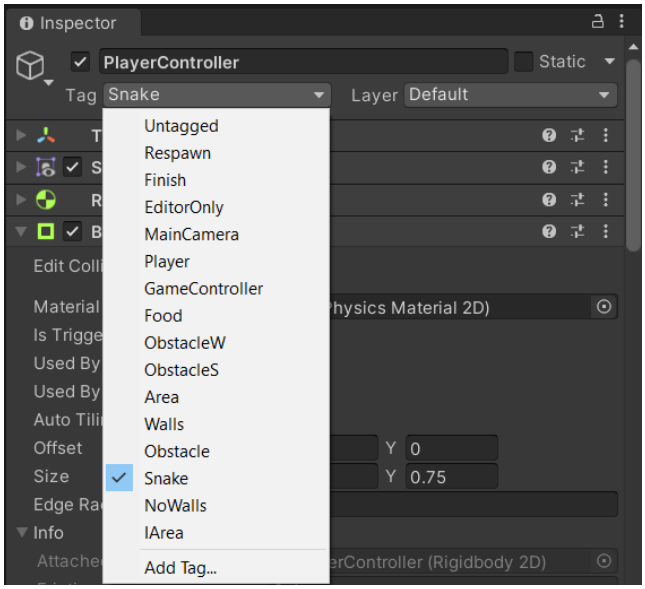
```
4 references
public void RandomPos()
{
    // Grab the bounds of the Grid Area and get random x and y values from these bounds
    Bounds bounds = gridArea.bounds;
    float x = Random.Range(bounds.min.x, bounds.max.x);
    float y = Random.Range(bounds.min.y, bounds.max.y);

    // Set the positon of the food to the x and y values obtained
    this.transform.position = new Vector3(Mathf.Round(x), Mathf.Round(y), 0.0f);
}
```

This function is not only called on the Awake function, however. Whenever the player collides with the food, the food object also needs to move to a random location.

Whenever the box collider in the food GameObject is collided by a GameObject that has a BoxCollider2D as a trigger, the function OnTriggerEnter2D will be called on both objects. In the case of the food, we can use this function to call *RandomPos* again whenever the food gets collided by a snake head:

```
// Calls when the food collider hits another box collider
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D other)
{
    // Tell the player that they have collided with something
    Debug.Log("collided");

    // Only run this part if the food has collided with a snake head
    if (other.tag == "Snake")
    {
        Instantiate(PickupEffect, transform.position, transform.rotation);
        RandomPos();
    }
}
```

However, as noticed above, we only want the food to change positions when it has been collided with a snake and nothing else. We do this by checking if the tag of the GameObject that collided with it is “Snake”. Tags are a way of identifying certain objects/groups of objects and they can be assigned (as a string) to GameObjects via the Unity Editor as such:



From here, the code can now check the tag of the object it collided with, and if it is of tag “Snake”, it can call the *RandomPos* function.

If this condition is satisfied the food will also instantiate a new GameObject *PickupEffect* at its position before randomly moving. This GameObject holds a particle system, which produces a visual effect that only the local player will see:

VIDEO HERE PLSSSSSSSSSS

The player itself also has to react to collecting food – specifically growing by one segment and adding one to their score (or *FoodCount*). We use the same *OnTriggerEnter* function (now on the *PlayerController* script) to detect whenever the player collides into something, and use the same method of checking if the GameObject has the tag “Food”. We also need to add a check to see if the snake colliding with the food is from the local player or not to remove the chance of another player influencing the score and snake length of the local player:

```
// Calls when the Snake Head collider hits another box collider
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D other)
{
    // Do not do anything if the Snake head is not from the local player
    if (!PV.IsMine)
    {
        return;
    }

    // If the snake collides with a food GameObject:
    if (other.tag == "Food")
    {
        // Call an RPC to grow the snake and increment the FoodScore
        PV.RPC("RPC_Grow", RpcTarget.All);
    }
```

If the object the snake collides with is indeed food, an RPC called *RPC_Grow* is called. RPCs (Remote Procedure Calls) are essentially function calls on all remote clients. When one client calls an RPC, every client who has the type of GameObject that called the RPC will execute it on their end. When calling an RPC, the client has to refer to their Photon View (the PV in *PV.RPC*), and must pass two arguments before they can pass any more – the first is the name of the RPC as a string, and the second is the target(s) of this RPC call.

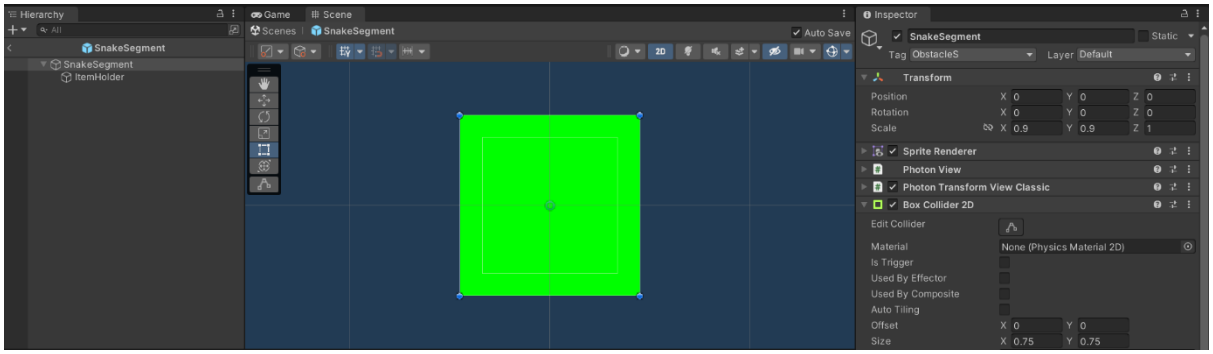
An RPC can target differing groups of players depending on what is specified. For example, *RpcTarget.All* sends the call to every client that has the type of GameObject that called the RPC, and *RpcTarget.Others* does the same but excludes the client sending the call. RPCs can also be sent to specific players by getting their PhotonViews and then getting their Player variable (which essentially holds all information about the player), before using that variable as the target instead of *RpcTarget. Targets* can also be buffered by the master server, so that players that join in the middle of a game can still get the same RPC calls as those who joined when the game started. However, as players will not be joining once a game has started, buffering targets is not needed. In this case, I will simply use *RpcTarget.All* as my target.

RPC_Grow’s main function is to increment the score of the local player, growing the snake by one segment, as well as syncing this growth to all other players in the room. To indicate to the program that this function is indeed an RPC, a tag [PunRPC] needs to be added above it:

```
[PunRPC]
0 references
private void RPC_Grow()
{
    // Do not run this function if the player running this is not the local player
    if (!PV.IsMine)
        return;

    // Wait for 0.02 seconds and then instantiate a snake segment to the position of the last item in the _segments list
    StartCoroutine("Wait", 0.02f);
    GameObject SnakeS = PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs", "SnakeSegment"), _segments[_segments.Count - 1].position, Quaternion.identity);

    // Add this segment to the list of segments
    Transform segment = SnakeS.transform;
    _segments.Add(segment);

    // Increment the FoodCount by a set amount and allow the player to change speed
    FoodCount += IncreaseAmt;
    ScoreText.text = FoodCount.ToString("00"); // Display the new score of the player to its UI element
    IsSpeedable = true;
}
```

In this function, we first check to see if the PhotonView of the player calling this function is that of the local player or not. This is to ensure that when one player collects food, every other player does not also undergo this function at the same time. Then, the program waits for a set amount of time before instantiating (via Photon) a prefab for a snake segment, set to the position of the last transform in the _segments list of transforms. This segment's position is then added to the *_segments* list and the score of the local player is incremented by an integer variable *IncreaseAmt*, currently set to 1. We finally display the current score of the player to the UI element we made beforehand holding the player's score.

The snake segment prefab we are instantiating in this RPC will be separate from the *PlayerController* prefab, as we will need to instantiate multiple of these as the player collects more food, but will still be a Photon Prefab (and will go in the Resources folder) to ensure that it is instantiated across the network. The prefab will look very similar to the Snake's head in the *PlayerController*, but will be slightly smaller to visually differentiate it from the head of the snake. It will also have a box collider, which will be used later on:



Notice in *RPC_Grow* how a Boolean variable *IsSpeedable* is set to true. This is because one of the features of this game is to speed up the player's snake for every 5 pieces of food it picks up. This will be done within the *Movement* Coroutine, as it relates to the movement of the player's snake itself.

```
        // If the player's snake is speedable and its score is a multiple of 5 greater than 0 run the following code:
        if (FoodCount > 0 && FoodCount % 5 == 0 && IsSpeedable)
        {
            // Reduce the amount of waiting time between moving the snake one unit by a factor of 1.25
            repeat_time = repeat_time / 1.25f;
            Debug.Log("Speed Up");
            IsSpeedable = false; // Ensure that the snake doesnt infinitely speed up
        }
        yield return new WaitForSeconds(repeat_time);
    }
}
```

Every time the snake moves, this section of the coroutine checks if the players *FoodCount* is a multiple of 5 greater than 0 and the *IsSpeedable* Boolean is true. If both of these conditions are satisfied, the time spent waiting between each movement of the snake is reduced by a factor of

1.25, and *IsSpeedable* is set to false, to ensure that when the player reaches a score that is a multiple of 5, it only speeds up once and not constantly.

Going back to *RPC_Grow*, a coroutine is started here named Wait, which essentially allows it to wait for a set amount of time before continuing with the code:

```
// Allows any function to wait for a specified time before continuing
0 references
IEnumerator Wait(float time)
{
    Waiting = true;
    yield return new WaitForSecondsRealtime(time);
    Waiting = false;
}
```

In addition, this RPC changes the UI element holding the score of the local player (that we created before this section). In order to do this, we create a Textbox variable holding the UI textbox we made, and then set the text of that textbox to the current *FoodCount* of the player, converted to a string.

```
ScoreText.text = FoodCount.ToString("00"); // Display the new score of the player to its UI element
```

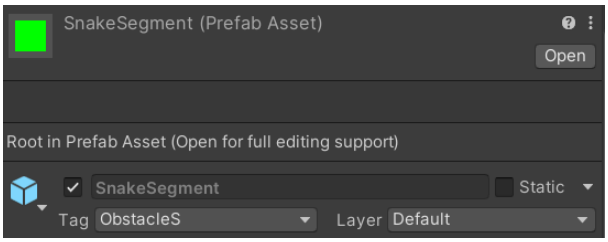
However, the canvas we put this UI element in should only be changed by the local player and nobody else. This is why, in the Start function, we say that if the player running the Start function is not the local player, remove that player's UI canvas. This way, the only canvas left over for each player is their own:

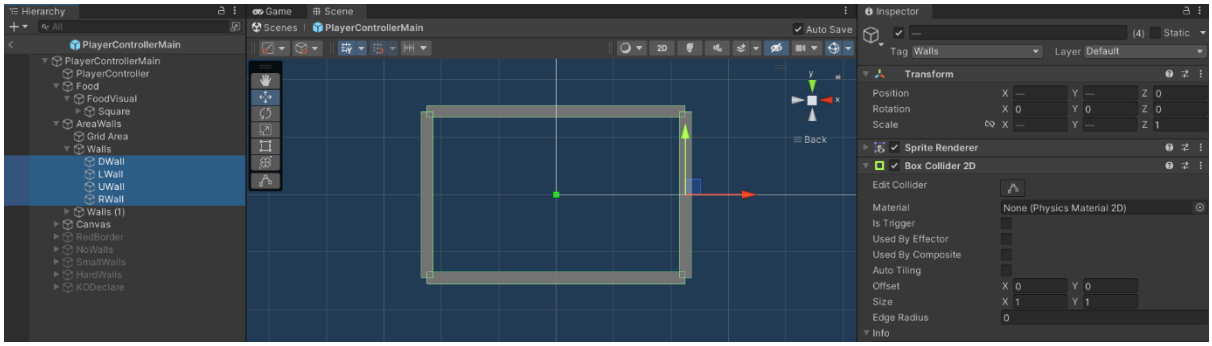
```
Unity Message | 0 references
void Start()
{
    // If this isn't the local player, dont run anything and destroy the player's UI canvas
    if (!PV.IsMine)
    {
        Destroy(uiCanvas);
        return;
    }
```

Player Death and Winning

Food is not the only GameObject a player's snake head may collide with in a game. It may also collide with the walls it is contained in, or a segment of itself. In either of these cases, a different function needs to be executed to handle the "death" of the player.

To start off, I tag each of the 4 *AreaWalls* sections and the *SnakeSegment* prefab with "Walls" and "ObstacleS" respectively in order to differentiate them from other tagged objects.





Then, in the *PlayerController* script, I will add to the *OnTriggerEnter2D* callback function code to run a different function when the snake collides with either a wall or itself:

```
// Otherwise if the snake collides with either the walls or itself:
else if ((other.tag == "Walls" || other.tag == "ObstacleS"))
{
    while (!IsLost) // While the Player has not lost yet:
    {
        PV.RPC("RPC_ShowKO", RpcTarget.All); // Call the RPC to show the K.O interface to all players in the room
        StopAllCoroutines(); // Stop all Coroutines currently active
        Die(); // Run the Die function (for the local player only)
        PV.RPC("RPC_Die", RpcTarget.All); // Run the Die RPC (for all players to see)
    }
```

Within this else if statement a few new functions and variables are used. An RPC named *RPC_ShowKO* is called, which enables the GameObject holding the visual showing a player has died for all players to see:

```
[PunRPC]
0 references
private void RPC_ShowKO()
{
    // Set the GameObject showing the K.O interface to be active
    KOIndicator.SetActive(true);
}
```

This visual is simply an empty GameObject holding an Image Object with the visual on it. The empty GameObject has a Photon View as well as a Photon Transform View to sync position to all other players in the room. This visual will also be disabled at default as we do not want to see the K.O visual when a player is first spawned in:

Back to the else if statement, a function *StopAllCoroutines* is called after the first RPC. Simply put, it automatically halts all running coroutines (such as *Movement* and *Direction*), ensuring the player does not have the ability to function any further.

After *StopAllCoroutines* is called, a local function *Die* is called. This runs only on the local client, so no other player will run this:

```
1 reference
private void Die()
{
    // Set the bool checking if the player has lost to true
    this.IsLost = true;

    // Notify all players that this player has died
    PVInstances = PlayerPVScript.GetPlayerViewList();
    foreach (PhotonView pv in PVInstances)
    {
        if (pv.ViewID != PV.ViewID)
        {
            Player player = pv.Controller;
            pv.RPC("RPC_PlayerDied", player, TargettedBy);
        }
    }

    // Activate the Game Over UI and the K.O text with that UI
    SetUIActive(1);
    KOImage.SetActive(true);

    // Display the number of kills a player got as well as their place in the room
    KillsText.text = Kills.ToString("00");
    PlaceText.text = Place.ToString("00");
    PlayerNumText.text = PhotonNetwork.CurrentRoom.PlayerCount.ToString("00");
}
```

First off in this function, the Boolean *IsLost* is set to true, indicating that the player has indeed lost. Then, a list of all player Photon Views are collected via a function in another script also in the *PlayerController* prefab (this will be further elaborated upon in the targeting section), and the code will increment through each player in that list, calling an RPC that the local player died to that player specifically if they are not the local player (by comparing the ViewIDs of the incremented player to the local player's ViewID).

This RPC, named *RPC_PlayerDied* has one argument; an integer showing a player's ID. This RPC calls a local function *PlayerDied* with the same ID argument, but only if the player receiving this call is the local player, and has not lost yet:

```
[PunRPC]
0 references
private void RPC_PlayerDied(int ID)
{
    // Call the PlayerDied function if the player is the local player and has not lost yet
    if (this.IsLost) { return; }
    else if (this.PV.IsMine)
    {
        PlayerDied(ID);
    }
    else
    {
        return;
    }

}
```

The local function *PlayerDied* is responsible for reducing an integer - indicating a player's placement - of every player in the room by 1 whenever a player has died, as well as incrementing a player's kill-count if they are the player that killed the player who died (this will be explained further when talking about kills in more detail later on). In addition, if the place that the player calling this function is at is equal to 1 (meaning that there is only one player left alive in the room), a separate local function named *Win* is called.

```
1 reference
private void PlayerDied(int ID)
{
    // Reduce this player's current placement in the room
    Place -= 1;

    // If this player killed the player who died, add one to their killcount
    if (ID == PV.ViewID)
    {
        Kills += 1;
    }

    // If this player is the one player left alive, call the Win function
    if(Place == 1)
    {
        Win();
    }
}
```

The *Win* function acts much like the *Die* function, but does not need to send an RPC to other players to change any of their variables, and activates the *Win* indicator (which is exactly like the K.O indicator except for the image it contains):

```
1 reference
private void Win()
{
    // Stop all Coroutines currently active
    StopAllCoroutines();

    // Activate the Game Over UI and the Win text with that UI
    SetUIActive(1);
    WinImage.SetActive(true);

    // Call the RPC to show the Win interface to all players in the room
    PV.RPC("RPC_ShowWin", RpcTarget.All);

    // Display the number of kills a player got as well as their place in the room
    KillsText.text = Kills.ToString("00");
    PlaceText.text = Place.ToString("00");
    PlayerNumText.text = PhotonNetwork.CurrentRoom.PlayerCount.ToString("00");
}
```

Part of this *Win* function is displaying UI telling the player locally they have won the game. This UI is made in the Unity Editor, with different UI GameObjects holding the general Game Over UI, as well as specific Won/Loss images:

The light green, dark green and white numbers are the *KillsText*, *PlaceText* and *PlayerNumText* variables respectively, and hold the numbers for *KillCount*, *Placement* and *Total* number of players in a room respectively. All 3 will be updated whenever a player either dies or wins, but the UI these three numbers are held in will only be displayed when the function *SetUIActive* is called with value 1.

```
3 references
public void SetUIActive(int index)
{
    // Set every UI GameObject in the UIParts list to inactive and only activate the UI gameobject of the specified index in UIParts
    foreach (GameObject ui in UIParts)
    {
        ui.SetActive(false);
    }
    UIParts[index].SetActive(true);
}
```

What this function does is disables all of the UI GameObjects for the player, and then only enables the UI elements that are in the specified index of *UIParts*, a list of UI GameObjects. This list is added to within the Unity Editor with both the In-Game UI, as well as the *GameOver* UI.

As the *GameOver* UI is element 1 in this list, *SetUIActive(1)* will enable that UI for the player to see.

The local Die function also enables this *GameOver* UI as well as display the *KillCount*. *PlaceText* and *PlayerNumText* to the player, but displays the “K.O!” image instead of the “You Won!” image.

In both cases (either a win or a loss), two buttons appear at the bottom of the screen for the player to click. The functions of these will be further explained within the “Spectating and Exiting the Game” section later on.

The last line of the else if statement checking if a player has collided with an obstacle calls an RPC named *RPC_Die*, with the target of all players. The main function here is the removal of all existing

segments of the snake and the syncing of this to all players in the room. This is done by going through each transform in the *_segments* list (and thus every snake segment in the game, from all players), and, if the segment in that transform is from the local player, removing that segment's transform from the *_segments* list and destroying that segment's GameObject via Photon:

```
[PunRPC]
0 references
private void RPC_Die()
{
    // Do not destroy the player's own snake if they are not the player who died
    if (!PV.IsMine)
        return;

    // Go through every transform in _segments
    for (int i = 1; i < _segments.Count; i++)
    {
        // Grab the Photon View of the segment
        GameObject s = _segments[1].gameObject;
        PhotonView sPV = s.GetPhotonView();

        // If the segment is from the player who died, destroy the segment
        if (sPV.IsMine)
        {
            _segments.Remove(s.transform);
            PhotonNetwork.Destroy(s);
        }
        else
        {
            continue;
        }
    }
```

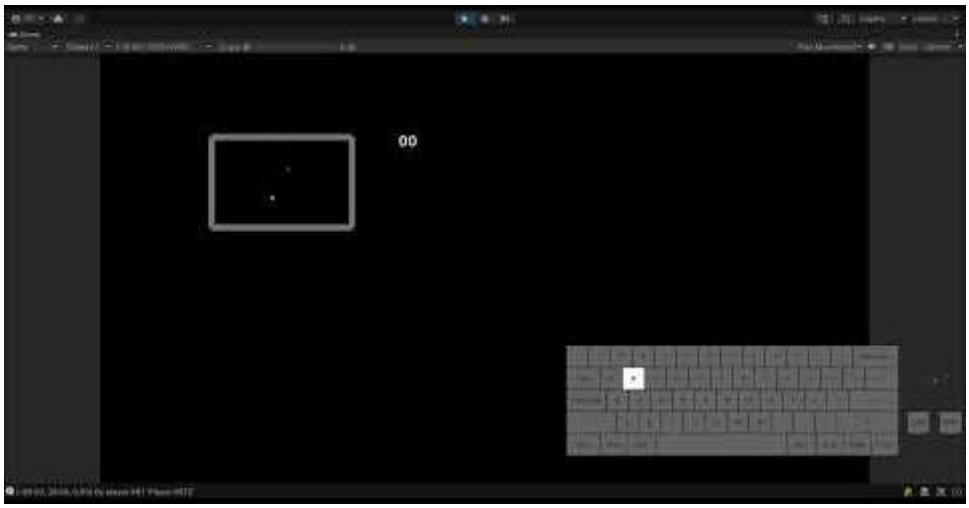
Testing current project

Tests in this section are done to ensure the snake moves and changes direction as planned, the food randomises its position and the process of colliding with both food and any obstacles call the correct functions to process player death.

In terms of movement, the following tests are done:

Test	**Expected Output**	**Actual Output**
S key pressed	Snake starts moving down	*PASS*
A key pressed	Snake starts moving left	*PASS*
W key pressed	Snake starts moving up	*PASS*
D key pressed	Snake starts moving right	*PASS*
S key pressed while moving up	Snake does not change direction	*PASS*
A key pressed while moving right	Snake does not change direction	*PASS*
W key pressed while moving down	Snake does not change direction	*PASS*
D key pressed while moving left	Snake does not change direction	*PASS*

A video showing these movement tests can be seen here:



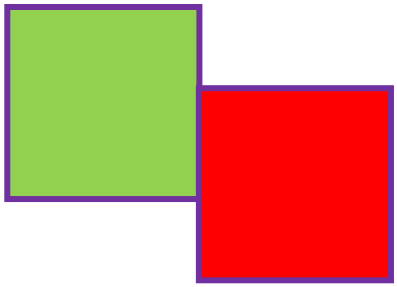
As is shown above, the snake seems to move in the direction inputted by the user (via the WASD keys), and the snake is not able to move backwards on itself with the opposite key press.

Now, collision testing is as follows:

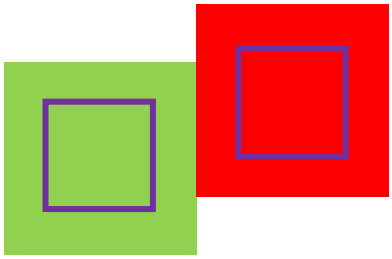
Test	**Expected Output**	**Actual Output**
Player collides with food	Player grows by 1 segment, FoodCount and Score UI increments by 1	Player immediately dies, Game Over UI appears
Player collides with walls	Player Dies, Game Over UI appears	*PASS*, but player dies before they actually go into the wall
Player runs along the side of a wall	Player will not die, snake continues moving	Player still dies before being able to run along the side of the wall
Player runs along the side of the food	Nothing happens, snake continues in same direction	Player dies, Game Over UI appears
Player collides with itself	Player Dies, Game Over UI appears	Unable to conduct, player dies when collecting food
Player runs along itself	Nothing happens, snake continues in same direction	Unable to conduct, player dies when collecting food

A video showing these collision tests can be seen here:

As is shown, whenever the snake gets close to any object with a collider, it triggers the *OnTriggerEnter2D* function, even when it doesn't actually run into it. After doing some research via forums, I realised that even the edges of two colliders hitting one another is considered a collision.

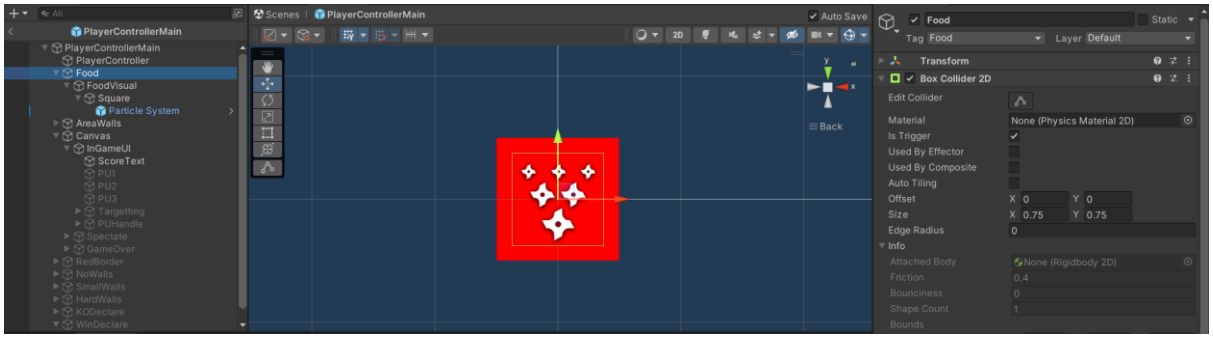


In both cases, even when the snake and the food do not actually run into each other, their colliders (in purple) still connect and thus trigger the function.



A solution to this is to simply reduce the size of the colliders on these objects, such that even when they run along each other, the colliders do not interact, thus not triggering *OnTriggerEnter2D*. This will not affect the function in general, as the snake still moves in increments, so if on one increment the snake is right next to the food, as long as it is facing the food, in the next increment it will still collide with the food's collider.

Adjusting the colliders of the food, snake and snake segments was done in the editor, by changing the size of the colliders in both x and y dimensions to 0.75:



With this change implemented, the tests, repeated, went as follows:

Test	**Expected Output**	**Actual Output**
Player collides with food	Player grows by 1 segment, FoodCount and Score UI increments by 1	Player immediately dies, Game Over UI appears
Player collides with walls	Player Dies, Game Over UI appears	*PASS*
Player runs along the side of a wall	Player will not die, snake continues moving	*PASS*

Player runs along the side of the food	Nothing happens, snake continues in same direction	*PASS*
Player collides with itself	Player Dies, Game Over UI appears	Unable to conduct, player dies when collecting food
Player runs along itself	Nothing happens, snake continues in same direction	Unable to conduct, player dies when collecting food

A video of this can be seen here:

As seen above, while the snake is now able to run along objects without triggering *OnTriggerEnter2D*, and the snake is now able to collide with walls and die, colliding with food still forces the snake to die.

Upon using breakpoints to find when exactly in the code the player dies, I found out that when the player runs into food, the snake segment spawns initially on top of the player, causing it to run into its own segment, killing it.

The solution to this was to essentially stop the *movement* coroutine the player has running until the snake segment has fully been instantiated, and then start the player moving again:

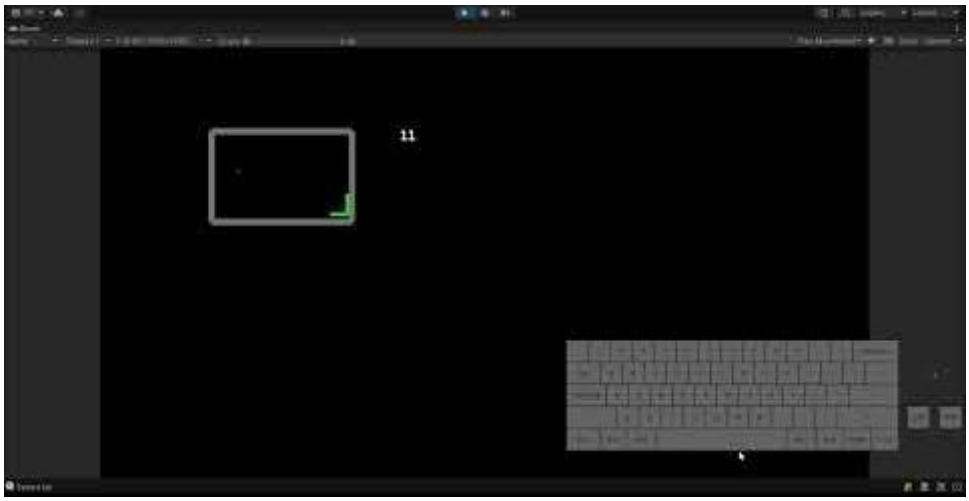
```
// If the snake collides with a food GameObject:
if (other.tag == "Food")
{
    StopCoroutine("Movement");
    PV.RPC("RPC_Grow", RpcTarget.All);
    StartCoroutine("Movement");
```

Now, running the same tests one more time:

Test	**Expected Output**	**Actual Output**
Player collides with food	Player grows by 1 segment, FoodCount and Score UI increments by 1	*PASS*
Player collides with walls	Player Dies, Game Over UI appears	*PASS*
Player runs along the side of a wall	Player will not die, snake continues moving	*PASS*
Player runs along the side of the food	Nothing happens, snake continues in same direction	*PASS*

Player collides with itself	Player Dies, Game Over UI appears	*PASS*
Player runs along itself	Nothing happens, snake continues in same direction	*PASS*
Player collects a multiple of 5 number of food	The player speeds up every multiple of 5	*PASS*

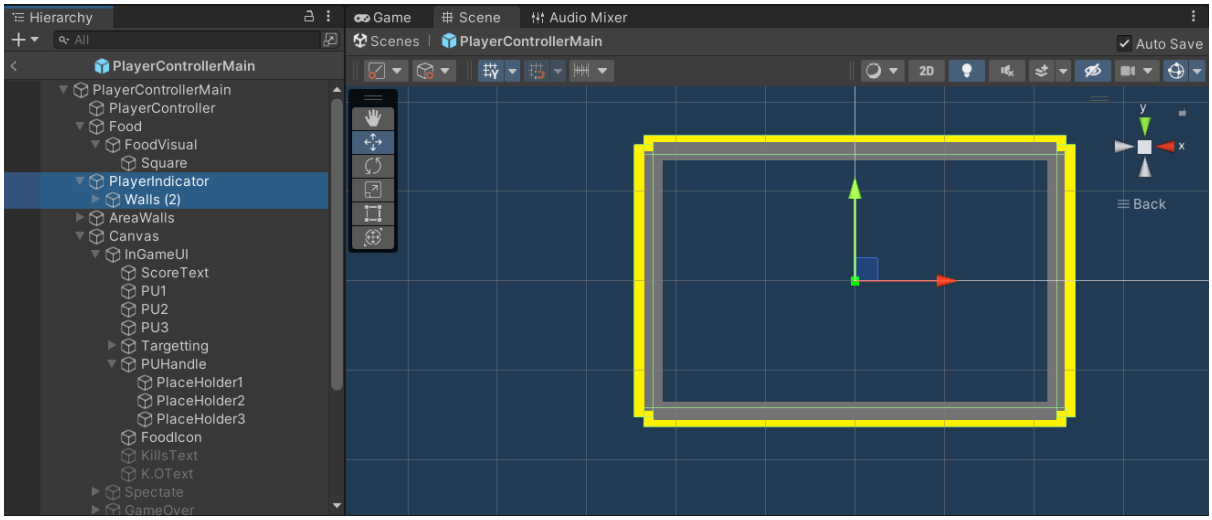
A video of this can be seen here:



Extra consideration

Some extra consideration here is that the player may not even know which snake they are controlling, as all play areas look the same.

To fix this, I add a yellow outline to the *PlayerController* prefab as such:



In the *PlayerController* script, I ensure this only shows for the local player by destroying all instances of this outline (named indicator) that are not from the local player:

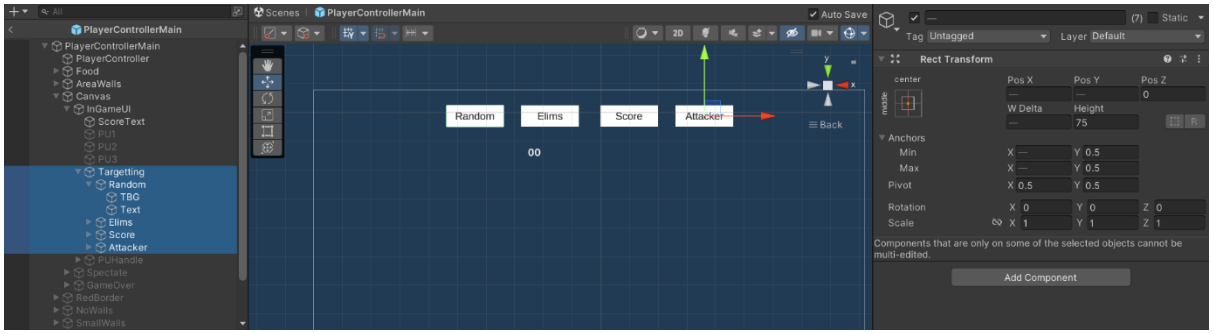
```
Unity Message | 0 references
void Start()
{
    // If this isn't the local player, dont run anything and destroy the player's UI canvas
    if (!PV.IsMine)
    {
        Destroy(uiCanvas);
        Destroy(indicator);
        return;
    }
```

Targeting

This section of development is dedicated to the act of targeting other players in a room.

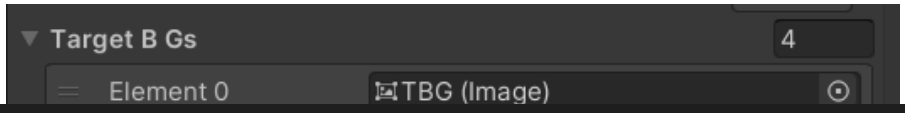
Targeting Methods

In the game, the player should be able to target 4 kinds of players. As such, there should be 4 UI elements the player can switch between in order to choose who to target. For each form of targeting, a UI element is made that holds a placeholder background and text:



Back in the *PlayerController* Script, I will assign references to the backgrounds these UI elements as a list of Images, and set an integer to check what targeting element the user is currently pointing at:

```
[SerializeField] List<Image> TargetBGs;
private int BGListPointer;
```




```
// Calls when target needs to be changed (on user input)
2 references
private void TargetIncrement()
{
    // Increment BGListPointer, lopping it back to 0 when it goes over 3
    BGListPointer = (BGListPointer + 1) % 4;

    // Reset the alpha of all targetting UI element images to dim them slightly
    foreach (Image i in TargetBGs)
    {
        var tempColour = i.color;
        tempColour.a = 0.3f;
        i.color = tempColour;
    }

    // Set the alpha of the targetting UI element in the index of BGListPointer to 1, highlighting it
    Image tempImage = TargetBGs[BGListPointer];
    var tempColor = tempImage.color;
    tempColor.a = 1f;
    tempImage.color = tempColor;
}
```

When we want to switch the method of targeting (on a button press), we can call a function named *TargetIncrement*. This function increments the *BGListPointer* integer, ensuring that it is only between 0 and 3 (as there are only 4 elements in the *TargetBGs* list) with a modulo function. Then, it dims the images of all targeting UI elements before highlighting the one at the index provided by *BGListPointer*.

To call this function whenever a specific key is pressed, we can start a coroutine (named *TargetChange*) that can check if that key (in this case, P) is pressed. If so, it will run *TargetIncrement*, and if not, it will wait until the next frame to try again:

```
0 references
IEnumerator TargetChange()
{
    // Only do this for the local player
    while (PV.IsMine)
    {
        // If P is pressed, call TargetIncrement
        if (Input.GetKeyDown(KeyCode.P))
        {
            TargetIncrement();
        }
        // If not, wait until the next frame to check again
        yield return 0;
    }
}
```

We start this coroutine in the Start function, after calling *TargetIncrement* once already to ensure the *BGListPointer* is a positive value:

```
BGListPointer = -1;
TargetIncrement();

// Start the named coroutines
StartCoroutine("Movement");
StartCoroutine("Direction");
StartCoroutine("TargetChange");
```

Targeting Players

Depending on the method of targeting, different variables may be received and different functions may be called. We can change up what lines of code a player has to run by using a switch statement.

These work similar to if-else statements in the sense that they run differing lines of code depending on what condition is fulfilled. However, unlike if-else statements, these can only pass through one variable as a condition, but with the added benefit of being (typically) faster than their counterparts. This is because, unlike if-else statements, which essentially perform a lot of Boolean tests one at a time until they either reach a true test or the default branch, switch-case statements simply jump to the case corresponding to the value given to it. In this case, I will make a switch-case statement passing *BGListPointer*, as that is already a good enough variable to differentiate the forms of targeting.

Before I initiate the switch statement, however, I will need to find a way to get a list of all players in the room. I do this by calling a function named *GetPlayerViewList*. This function exists in a separate class named *PlayerPVScript*, which contains a static list Instances of Photon Views of every player in the room.

Here, the code makes itself a singleton as long as the player holding the script is the local player, and adds the player's Photon View to Instances. When the *PlayerController* holding this script is destroyed, the code also handles removing their Photon View from Instances.

```
public class PlayerPVScript : MonoBehaviour
{
    PhotonView photonView;
    public static List<PhotonView> Instances = new List<PhotonView>();
    public static PlayerPVScript LocalInstance;

    void Awake()
    {
        photonView = gameObject.GetComponent<PhotonView>();

        Instances.Add(photonView);

        if (photonView.IsMine)
        {
            LocalInstance = this;
        }
    }

    void OnDestroy()
    {
        Instances.Remove(photonView);
    }
```

In the *PlayerController* script, a new coroutine is added named *PlayerTarget* is made. This coroutine, after checking that the player running it is the local player, grabs the Instances list from *PlayerPVScript* as a new list variable *PVInstances*, and then initialises a value *MaxValue* as 0. Then, based on what the value *BGListPointer* took at that time, different code is executed.

```
IEnumerator PlayerTarget()
{
    while (PV.IsMine)
    {
        // Grab the Instances list from PlayerPVscript and set a MaxValue indicator to 0
        List<PhotonView> PVInstances = PlayerPVScript.Instances;
        int MaxValue = 0;

        switch (BGListPointer)
        {
            case 0: //Random Player
                // Choose a random Photon View from the Instances list
                PhotonView ChosenPV = PVInstances[Random.Range(0, PlayerPVScript.Instances.Count)];
                break;
```

If *BGListPointer* is 0, then simply choose a random Photon View from *PVInstances*.

```
case 1: //Max Score
    foreach (PhotonView pv in PVInstances)
    {
        // Iterate through every Player, calling RPC_CallToPlayer
        PV.RPC("RPC_CallToPlayer", pv.Controller, PV, 1);
        if (tempValue > MaxValue)
        {
            // If the value returned is larger than the current max value, make that value the max value and set PlayerIndex to that player's Photon View
            MaxValue = tempValue;
            PlayerIndex = PVInstances.IndexOf(pv);
        }
    }
    break;

case 2: //Max Kills
    foreach (PhotonView pv in PVInstances)
    {
        // Iterate through every Player, calling RPC_CallToPlayer
        PV.RPC("RPC_CallToPlayer", pv.Controller, PV, 2);
        if (tempValue > MaxValue)
        {
            // If the value returned is larger than the current max value, make that value the max value and set PlayerIndex to that player's Photon View
            MaxValue = tempValue;
            PlayerIndex = PVInstances.IndexOf(pv);
        }
    }
    break;
```

If *BGListPointer* is either 1 or 2, the code will increment through every Photon View in *PVInstances*, calling a new RPC *RPC_CallToPlayer* to that Photon View, passing through the local player's Photon View and the value of *BGListPointer*. If the value that is returned from this RPC (stored as an int *tempValue*) is larger than the current maximum value at the time, then that value becomes the new maximum value and an integer variable *PlayerIndex* takes the value of the index of the corresponding Photon View within *PVInstances*.

```
case 3: //Attacker
    foreach (PhotonView pv in PVInstances)
    {
        // Iterate through every Player, calling RPC_CallToPlayer
        PV.RPC("RPC_CallToPlayer", pv.Controller, PV, 3);
        if (tempValue == -10)
        {
            // If the value returned is -10, set PlayerIndex to that player's Photon View
            PlayerIndex = PVInstances.IndexOf(pv);
        }
    }
    break;
```

If *BGListPointer* is 3, the code will also iterate through every Photon View in *PVInstances*, calling *RPC_CallToPlayer* on each one. This time, however, this code only updates *PlayerIndex*, and only if the value that is returned is -10.

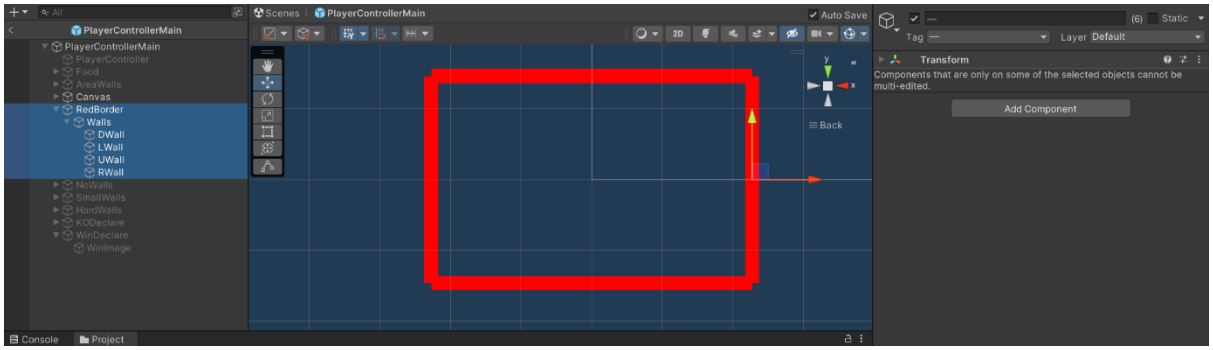
RPC_CallToPlayer is a function that takes arguments of type *PhotonView* and integer, and returns the *Food*, *KillCount*, or the value -10 based on what the value of the integer passed to it is:

```
[PunRPC]
private void RPC_CallToPlayer(PhotonView _pv, int _c)
{
    // Initialise an interger sentInt to 0
    int sentInt = 0;
    switch (_c)
    {
        case 3: // If the integer passed through was 3, send back to the player who sent the initial RPC call -10, only if this player is targetting that player
            sentInt = (PlayerPVScript.Instances[PlayerIndex] == _pv) ? -10 : 0;
            break;
        case 2: // If the integer passed through was 2, send back to the player who sent the initial RPC call the score this player has
            sentInt = FoodCount;
            break;
        case 1: // If the integer passed through was 1, send back to the player who sent the initial RPC call the number of kills this player has
            sentInt = Kills;
            break;
    }
    // Return whatever value was sent to the player who sent the inital RPC call
    PV.RPC("RPC_ReturnToPlayer", _pv.Controller, sentInt);
}
```

As this function is a void function (meaning it does not return a value), and it is running on a different client to the client that sent the RPC call, there needs to be a way to get the value of sentient back to the RPC caller. To do this, I make the client running this RPC call an RPC named *RPC_ReturnToPlayer* back to the client calling this RPC, simply setting the value it receives as the *tempValue* for that client.

```
[PunRPC]
private void RPC_returnToPlayer(PhotonView _pv, int _c)
{
    tempValue = _c;
}
```

Now that we have a way to find a target player, there still needs to be a way to signal to the player who they are targeting. As a visual indicator, I duplicated the *AreaWalls* GameObject of the *PlayerController* prefab, making it red as well as slightly larger than the *AreaWalls*. In addition, I disabled any box colliders as they are not needed for what is purely a visual indicator. I set this GameObject to disabled on default as the player does not need to see it as soon as they are spawned.



I will then assign this to the *PlayerController* script as a GameObject variable *BorderPrefab*. In the code for *RPC_ReturnToPlayer*, I will add an Vector3 argument _v and set a new Vector3 variable *TargetPosition* to _v:

```
[PunRPC]
private void RPC_returnToPlayer(PhotonView _pv, int _c, Vector3 _v)
{
    // Set the value to be checked back at PlayerTarget to _c and the Target's position to _v
    tempValue = _c;
    TargetPosition = _v;
}
```

In the *RPC_CallToPlayer* script, I will make a new local Vector3 variable *PlayerPosition* and set it to the *PlayerControllerMain* GameObject's (the parent of *PlayerController*) position. At the point where *RPC_ReturnToPlayer* is called, I will add *PlayerPosition* as the final value to pass into the RPC:

```
// Return whatever value was sent to the player who sent the inital RPC call
PV.RPC("RPC_ReturnToPlayer", _pv.Controller, sentInt, PlayerPosition);
```

In the *PlayerTarget* coroutine, I will add code to activate the *BorderPrefab* if it is disabled, and set its position to the current target's position. The coroutine will then wait a second before targeting again:

```
    // If the BorderPrefab is not already active, activate it
    if (!BorderPrefab.activeInHierarchy) { BorderPrefab.SetActive(true); }

    // Set the position of the BorderPrefab to the Target's position
    BorderPrefab.transform.position = new Vector3(
        TargetPosition.x,
        TargetPosition.y,
        1.0f
    );
    yield return new WaitForSeconds(1.0f);
}
```

Testing Current Project

Testing the project as it is may prove to be difficult given that the game might end by a player crashing into a wall before anything meaningful has happened. To prevent this, in the code depicting what the player does when colliding with an obstacle, everything except the call of *RPC_Die* is commented out, preventing those lines from running:

```
// Otherwise if the snake collides with either the walls or itself:
else if ((other.tag == "Walls" || other.tag == "ObstacleS"))
{
    while (!IsLost) // While the Player has not lost yet:
    {
        //PV.RPC("RPC_ShowKO", RpcTarget.All); // Call the RPC to show the K.O interface to all players in the room
        //StopAllCoroutines(); // Stop all Coroutines currently active
        //Die(); // Run the Die function (for the local player only)
        PV.RPC("RPC_Die", RpcTarget.All); // Run the Die RPC (for all players to see)
    }
```

In addition, within *RPC_Die*, extra lines are added to reset the player's score and speed, as well as moving the player to a random position within the walls.

```
FoodCount = 0;
ScoreText.text = FoodCount.ToString("00");
RandomPosSnake();
repeat_time = 0.1f;
```

RandomPosSnake simply imitates the process the food does to move around randomly:

```
1 reference
private void RandomPosSnake()
{
    if (!PV.IsMine)
    {
        return;
    }

    Bounds bounds = foodScript.gridArea.bounds;

    float x = Random.Range(bounds.min.x, bounds.max.x);
    float y = Random.Range(bounds.min.y, bounds.max.y);

    this.transform.position = new Vector3(Mathf.Round(x), Mathf.Round(y), 0.0f);
```

Now, testing was done as follows:

Test	Expected Output	Actual Output
Starting the game	Targeting UI is defaulted to Random	*PASS*
Random Targeting	The Red border highlights a random opponent	The local player is highlighted sometimes as well

In the case of random targeting, the red border can highlight the local player because there is no check to see if the Photon View that is randomly selected is the local player's own. Adding this check is done with a while statement, as such:

```
switch (BGListPointer)
{
    case 0: //Random Player
        // Choose a random Photon View from the Instances list, as long as it is not the local player
        PhotonView ChosenPV = PVInstances[Random.Range(0, PlayerPVScript.Instances.Count)];
        while (ChosenPV.isMine)
        {
            PhotonView ChosenPV = PVInstances[Random.Range(0, PlayerPVScript.Instances.Count)];
        }
        break;
```

Testing continues as follows:

Test	Expected Output	Actual Output
Pressing P	The form of targeting is changed, shown in the Targetting UI	*PASS*
Random Targeting	The Red border highlights a random opponent	*PASS*
Score Targeting	The Red border highlights the opponent with the highest score	An error is thrown

The error thrown in particular here is "An object reference is required for the non-static field, method, or property *'PlayerPVScript.photonView'*." and is thrown in the line where *RPC_CallToPlayer* is called. Checking with forums and documentation, this error is thrown because RPCs cannot take in Photon Views as an argument. To fix this, players need to be iterated though by another means, that being their ViewID. To fix this error, a static function is first made in the *PlayerPVScript* that returns its Instances list:

```
7 references
public static List<PhotonView> GetPlayerViewList()
{
    return Instances;
}
```

In addition, I will make a new integer variable *MyID* that holds the local player's own View ID, and have the *PVInstances* list call from *GetPlayerViewList* to grab the list of Photon Views:

Then, for every following part of code that references Photon Views directly (apart from iterating through *PVInstances*), replace the use of Photon Views with ViewIDs, and get the target to call the RPC to by referencing the incremented Photon View's controller:

```
0 references
IEnumerator PlayerTarget()
{
    while (PV.IsMine)
    {
        // Grab the list of Photon Views from PlayerPVScript
        PVInstances = PlayerPVScript.GetPlayerViewList();
```

(In addition, adding "*this*." Prefixes to certain variables endures that only the variables for that player specifically are referenced/changed. Also, any *Debug.LogError* statements are simply there as part of testing to check that the code returns something.)

```
case 2: //Max Score
    // Set a MaxValue indicator to 0
    int MaxValue2 = 0;
    foreach (PhotonView pv in PVInstances)
    {
        // Iterate through every Player using their ViewID, calling RPC_CallToPlayer
        Player player = pv.Controller;
        PV.RPC("RPC_CallToPlayer", player, pv.ViewID, MyID, 2);
        if (this.tempValue > MaxValue2)
        {
            // If the value returned is larger than the current max value, make that value the max value and set PlayerID to that player's ViewID
            MaxValue2 = this.tempValue;
            PlayerID = pv.ViewID;
        }
    }
    break;
case 3: //Attacker
    foreach (PhotonView pv in PVInstances)
    {
        // Iterate through every Player using their ViewID, calling RPC_CallToPlayer
        Player player = pv.Controller;
        PV.RPC("RPC_CallToPlayer", player, pv.ViewID, MyID, 3);
        if (tempValue == -10)
        {
            // If the value returned is -10, set PlayerID to that player's ViewID
            PlayerID = pv.ViewID;
        }
    }
    break;
```

```
1 reference
private void CallToPlayer(int CallID, int ReturnID, int _c)
{
    if (_c > 0)
    {
        // Set the ID of the player to return a value to as the ReturnID passed to it and set sentInt to 0
        int rID = ReturnID;
        sentInt = 0;

            if (_c == 3)
            {
                // If the integer passed through was 3, send back to the player who sent the initial RPC call -10, only if this player is targetting that player
                this.PVInstances = PlayerPVScript.GetPlayerViewList();
                this.sentInt = (this.PVInstances[PlayerID].ViewID == CallID) ? -10 : 0;
            }
            else if (_c == 2)
            {
                // If the integer passed through was 2, send back to the player who sent the initial RPC call the score this player has
                this.sentInt = this.FoodCount;
            }
            else if (_c == 1)
            {
                // If the integer passed through was 1, send back to the player who sent the initial RPC call the number of kills this player has
                Debug.LogError("sentInt = " + Kills);
                this.sentInt = this.Kills;
            }
    }

    // Return whatever value was sent to the player who sent the inital RPC call by finding that player's ViewID
    PVInstances = PlayerPVScript.GetPlayerViewList();
    foreach (PhotonView _pv in PVInstances)
    {
        if (_pv.ViewID == ReturnID)
        {
            Player player = _pv.Controller;
            PV.RPC("RPC_ReturnToPlayer", player, ReturnID, sentInt);
        }
    }
}
```

```
switch (BGListPointer)
{
    case 0: //Random Player
        // Choose a random Photon View ID from the Instances list, as long as it is not the local player
        PlayerID = PVInstances[Random.Range(0, PlayerPVScript.Instances.Count)].ViewID;
        while (PlayerID == PV.ViewID)
        {
            PlayerID = PVInstances[Random.Range(0, PlayerPVScript.Instances.Count)].ViewID;
        }
        break;
    case 1: //Max Kills
        // Set a MaxValue indicator to 0
        int MaxValue = 0;
        foreach (PhotonView pv in PVInstances)
        {
                // Iterate through every Player using their ViewID, calling RPC_CallToPlayer
                Player player = pv.Controller;
                PV.RPC("RPC_CallToPlayer", player, pv.ViewID, MyID, 1);
                if (tempValue > MaxValue)
                {
                    // If the value returned is larger than the current max value, make that value the max value and set PlayerID to that player's ViewID
                    MaxValue = tempValue;
                    PlayerID = pv.ViewID;
                }
        }
        break;
```

Testing the code this time:

Test	Expected Output	Actual Output
Pressing P	The form of targeting is changed, shown in the Targetting UI	*PASS*
Random Targeting	The Red border highlights a random opponent	*PASS*
Score Targeting	The Red border highlights the opponent with the highest score	No errors are thrown, but no player is targeted, no matter who has the highest score

The actual output here was not surprising by the time it was tested. In each call of RPCs in both *PlayerTarget* and *RPC_CallToPlayer*, I had used the local player's own ViewID instead of the ViewID of the player I am iterating to. For example, instead of:

```
foreach (PhotonView pv in PVInstances)
{
        // Iterate through every Player using their ViewID, calling RPC_CallToPlayer
        Player player = pv.Controller;
        PV.RPC("RPC_CallToPlayer", player, pv.ViewID, MyID, 1);
        if (tempValue > MaxValue)
        {
            // If the value returned is larger than the current max value, make that value the max value and set PlayerID to that player's ViewID
            MaxValue = tempValue;
            PlayerID = pv.ViewID;
        }
}
```

The correct code should be:

```
foreach (PhotonView pv in PVInstances)
{
        // Iterate through every Player using their ViewID, calling RPC_CallToPlayer
        Player player = pv.Controller;
        pv.RPC("RPC_CallToPlayer", player, pv.ViewID, MyID, 1);
        if (tempValue > MaxValue)
        {
            // If the value returned is larger than the current max value, make that value the max value and set PlayerID to that player's ViewID
            MaxValue = tempValue;
            PlayerID = pv.ViewID;
        }
}
break;
```

(For those struggling to find the difference, I changed *PV.RPC* to *pv.RPC* as the player I am iterating through has a reference to their Photon View as *pv*, while the local player's Photon View is referenced by PV.)

Repeating this for all the RPC calls in *PlayerTarget* and *RPC_CallToPlayer* and testing again, the following was found:

Test	Expected Output	Actual Output
Pressing P	The form of targeting is changed, shown in the Targetting UI	*PASS*
Random Targeting	The Red border highlights a random opponent	*PASS*
Score Targeting	The Red border highlights the opponent with the highest score	No errors are thrown, but the same player is targeted, no matter who has the highest score

The actual output here was not something I expected, but was a massive overlook on my part. RPC methods travel over a network, and thus have a certain latency to them; they aren't instant, but the code will continue on once it has been called as if it is.

To help fix this issue, I add a new Boolean variable called *IsChecked*, which is set to false before *RPC_CallToPlayer* is called.

```
case 1: //Max Kills
    // Set a MaxValue indicator to 0
    int MaxValue = 0;
    foreach (PhotonView pv in PVInstances)
    {
        // Iterate through every Player using their ViewID, calling RPC_CallToPlayer
        Player player = pv.Controller;
        IsChecked = false // Set IsChecked to false
        pv.RPC("RPC_CallToPlayer", player, pv.ViewID, MyID, 1);
```

Then, once the RPC has started calling, we use the line "*yield return new WaitUntil(() => IsChecked)*" to ensure that the line after this is only run when *IsChecked* is true.

```
yield return new WaitUntil(() => IsChecked); // Only continue with the code when IsChecked is true
if (this.tempValue > MaxValue)
{
    // If the value returned is larger than the current max value, make that value the max value and set PlayerID to that player's ViewID
    MaxValue = tempValue;
    PlayerID = pv.ViewID;
}
```

IsChecked is only made true in the *RPC_ReturnToPlayer* call, after a value has been obtained from the player being sent the RPC.

```
1 reference
private void ReturnToPlayer(int _c)
{
    this.tempValue = _c;
    this.IsChecked = true; // Set the local player's IsChecked to true
}
```

Testing the code with these edits produces the following:

Test	Expected Output	Actual Output
Pressing P	The form of targeting is changed, shown in the Targetting UI	*PASS*
Random Targeting	The Red border highlights a random opponent	*PASS*
Score Targeting	The Red border highlights the opponent with the highest score	*PASS,* But the player themselves can be targeted themselves

Elimination Targeting	The Red border highlights the opponent with the highest KillCount	*PASS,* But the player themselves can be targeted themselves
Attacker Targeting	The Red border highlights the opponent who is targeting them	*PASS*

WORK ON HERE

Power-Ups

Setting up Power-Ups

One of the defining features of this game is using various Power-Ups to either assist yourself or hinder an opponent. As the design section says, this is done by assigning certain button presses to functions that select a random power-up from different lists of power-ups. Then, pressing the same button again will trigger that power-up, which is either applied to the player themselves or the player they are currently targeting.

First off we will make the power-ups themselves, or at least information about them. Each Power-Up will be in its own class that will inherit from a parent class *PUItem*. As we need to allow this class to be inherited from, as well as being able to create objects from it, we replace the *MonoBehaviour* class inherit with *ScriptableObject*:

```
Unity Script | 10 references
public class PUItem : ScriptableObject // Allow this class to create objects and be inherited from
{
    // Initialise variables for Power-Up name, description, activation time, a Visual for the Power-Up, and a value to differentiate Power-Ups of a similar type
    public string Name;
    public string Description;

    public float ActivateTime;

    public Sprite Visual;

    public float Value;
}
```

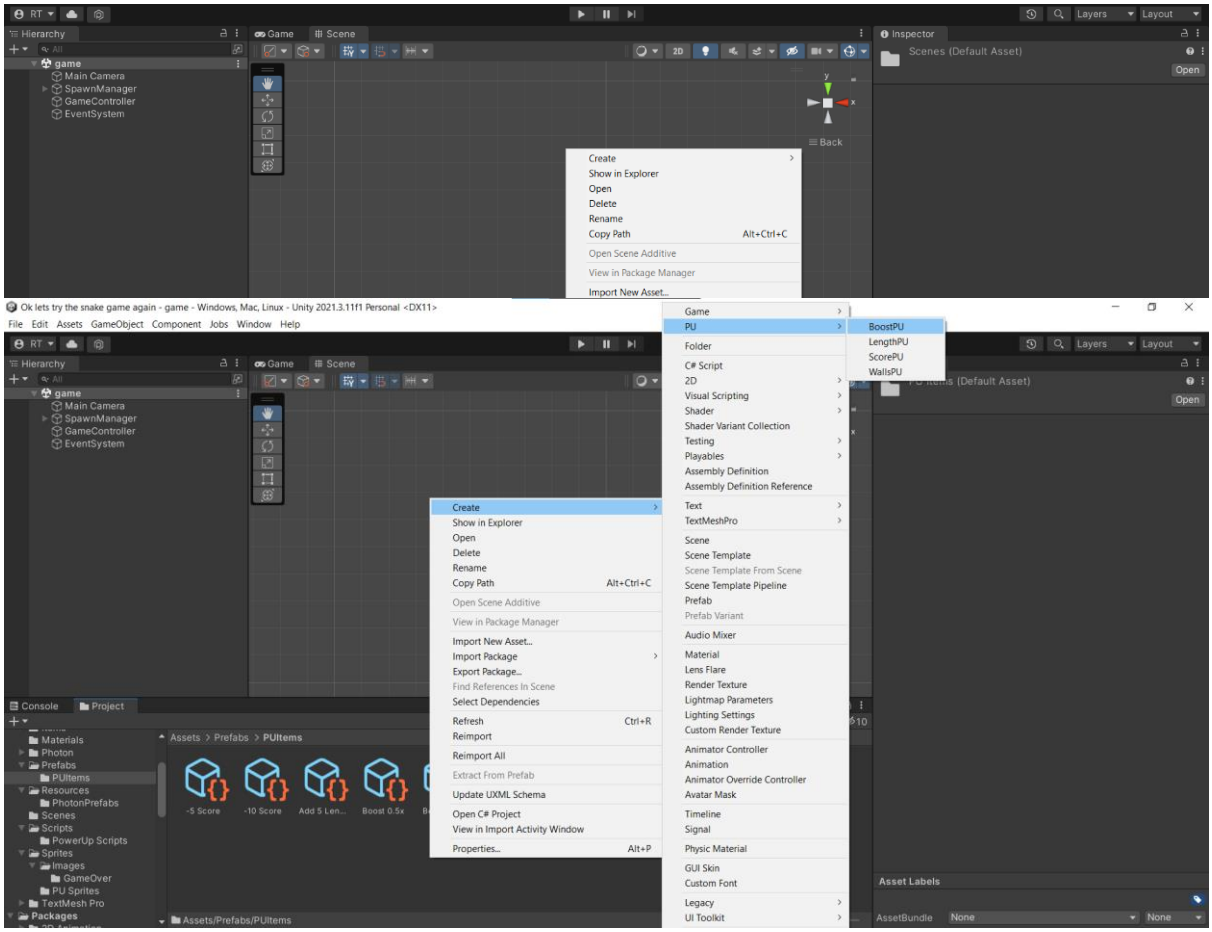
This is all we need for the base Power-Up class, and making classes for types of Power-Ups is done is not done with many more lines. There are 4 kinds of Power-Ups that are made here: Changing a player's speed (Boosting), changing a player's snake length, changing a player's score, and changing what walls a player uses.

For each type of Power-Up we will be making a class that inherits from *PUItem*. For example, with the boosting power-up type, we make a new script named *BoostPU*, and add these lines to it:

```
[CreateAssetMenu(menuName = "PU/BoostPU")] // Allow objects from this class to be created from the Unity Editor
Unity Script | 2 references
public class BoostPU : PUItem // Inherit from the base PUItem class
{

}
```

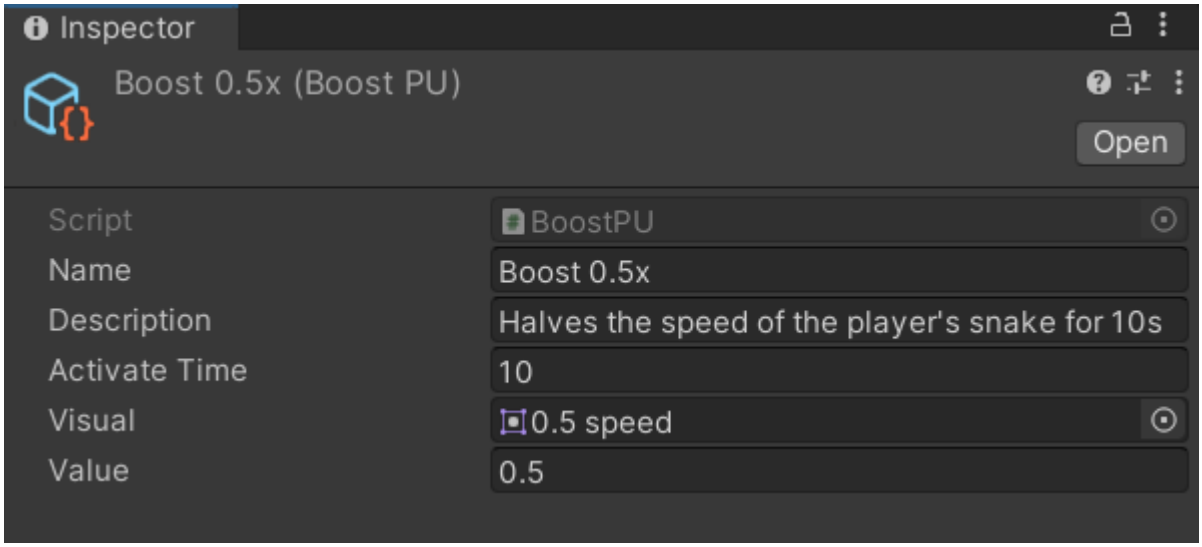
First, we make *BoostPU* inherit from *PUItem*, thus giving it its own Name, Description, etc from the base class. It also inherits the *ScriptableObject* class, allowing objects of *BoostPU* to be made. At the very top of the above image, a *CreateAssetMenu* function is called, which essentially allows for objects based on these classes to be created via the Unity Editor.

In this interface, right clicking on an empty space in the Project section brings up a menu allowing for you to perform multiple different functions relating to project files:

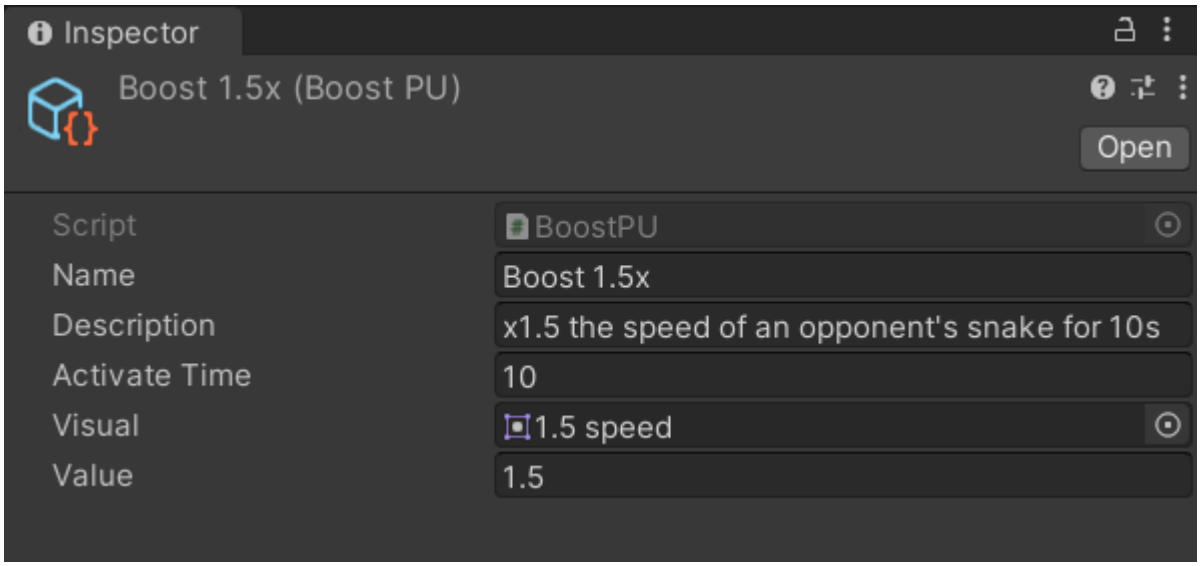


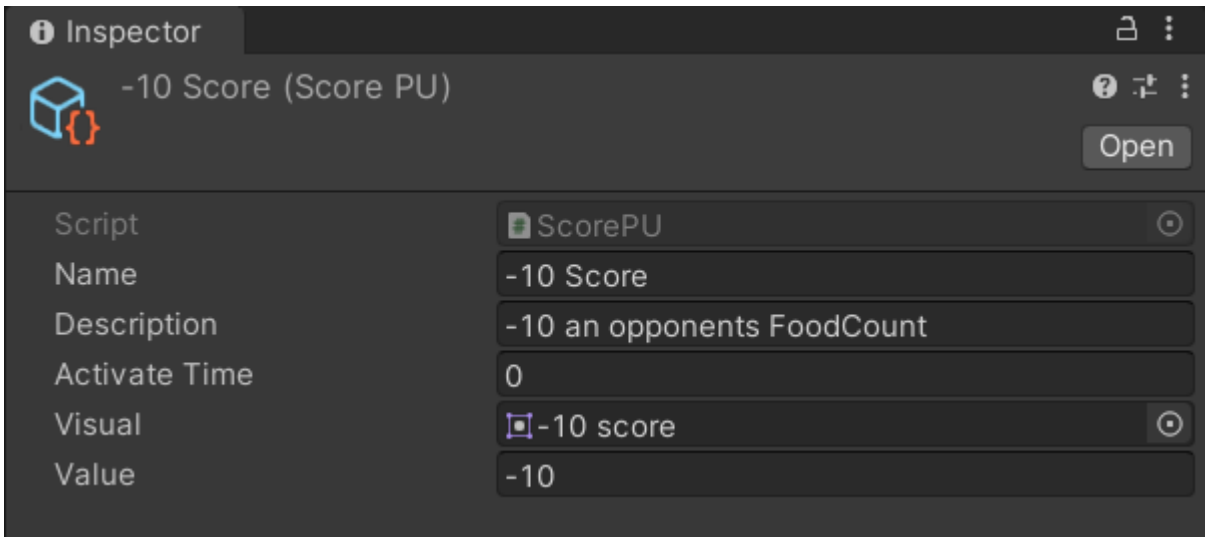
Typically, this menu can be used to add scripts, textures/materials, animation controllers, and more. However, due to the *CreateAssetMenu* function in the script for each power-up type, we can now also add objects relating to these classes. For example, there are 3 types of boost power-ups I want to make – one that halves the speed of the local player, one that multiplies the speed of an opponent by 1.5, and one that doubles an opponents speed. For each one, I can create a object of class *BoostPU* by following the right click menu through Create/PU/*BoostPU* (the menu name I specify within the *CreateAssetMenu* function).

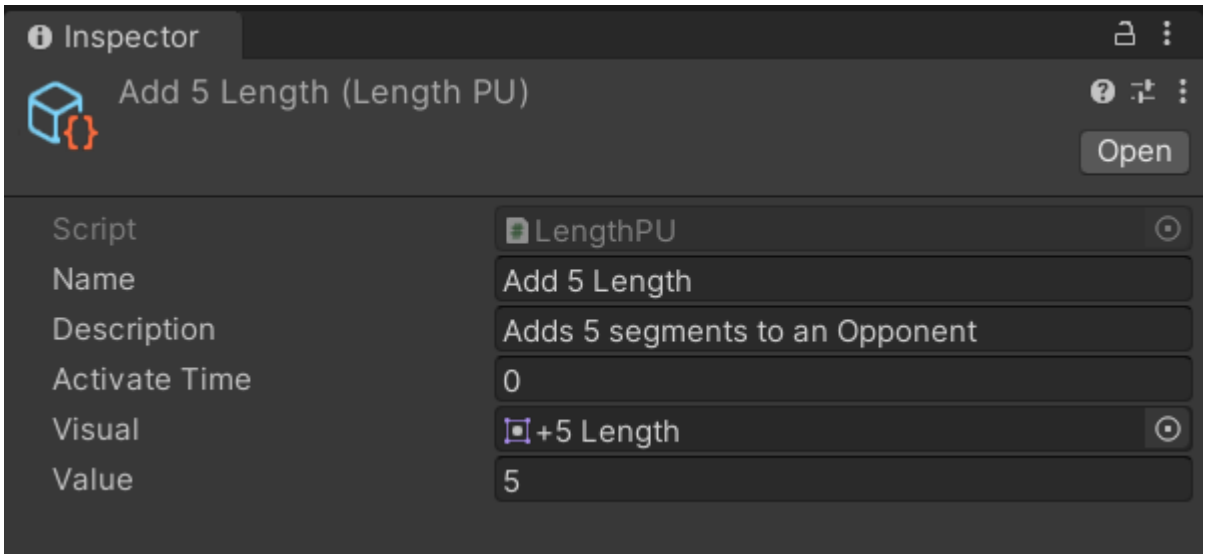
This makes a new empty object with values for Name, Description, etc that I can fill in within the editor. For the boost power-up that halves the local player's speed, I can put in the following details:

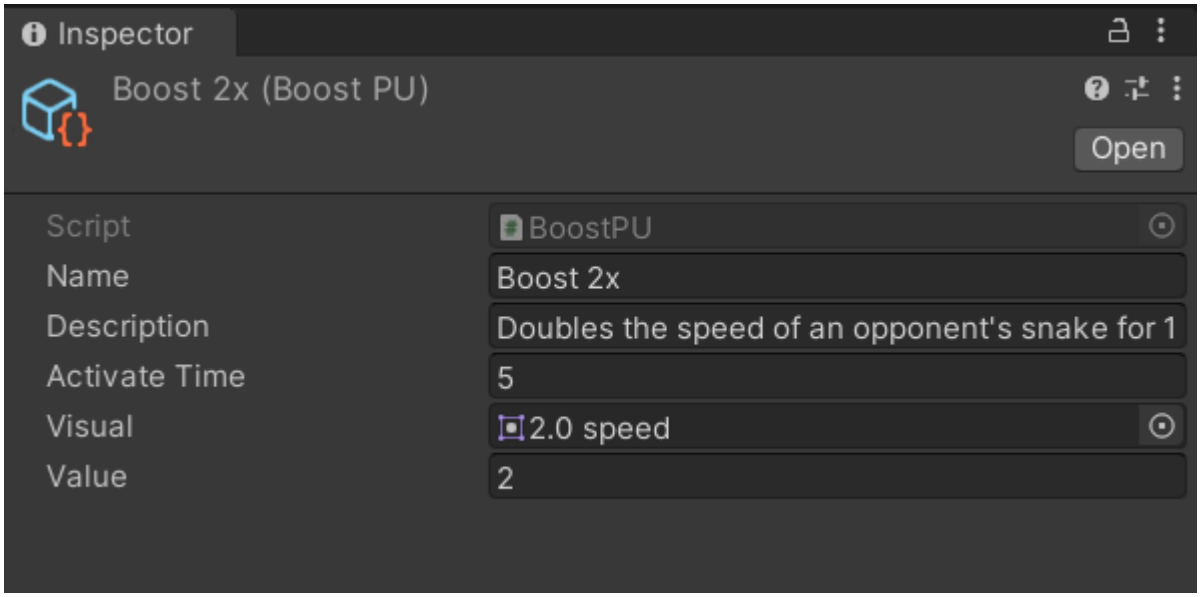


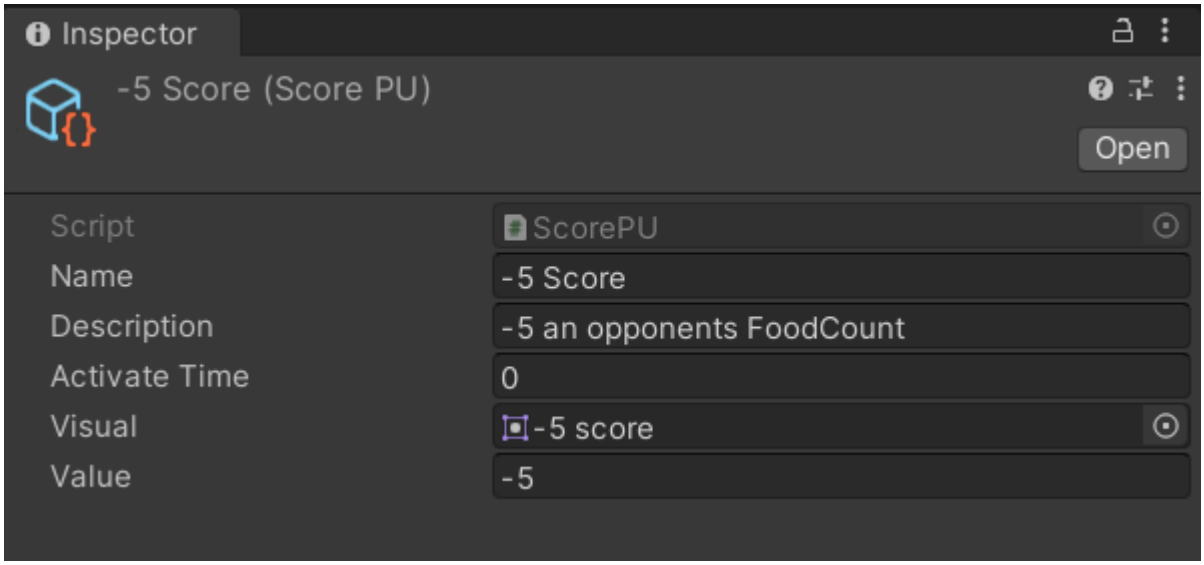
This is for one specific power-up however. Creating the other types of power-ups requires me to repeat this whole process of inheriting from *PUItem* and creating new objects. The rest of the power-ups are as follows:

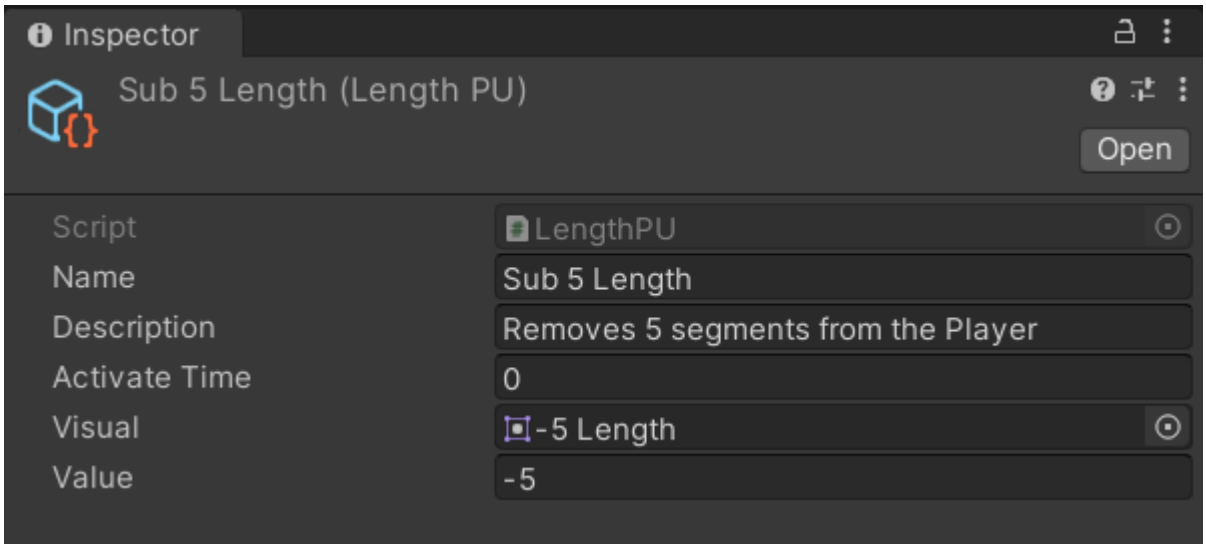


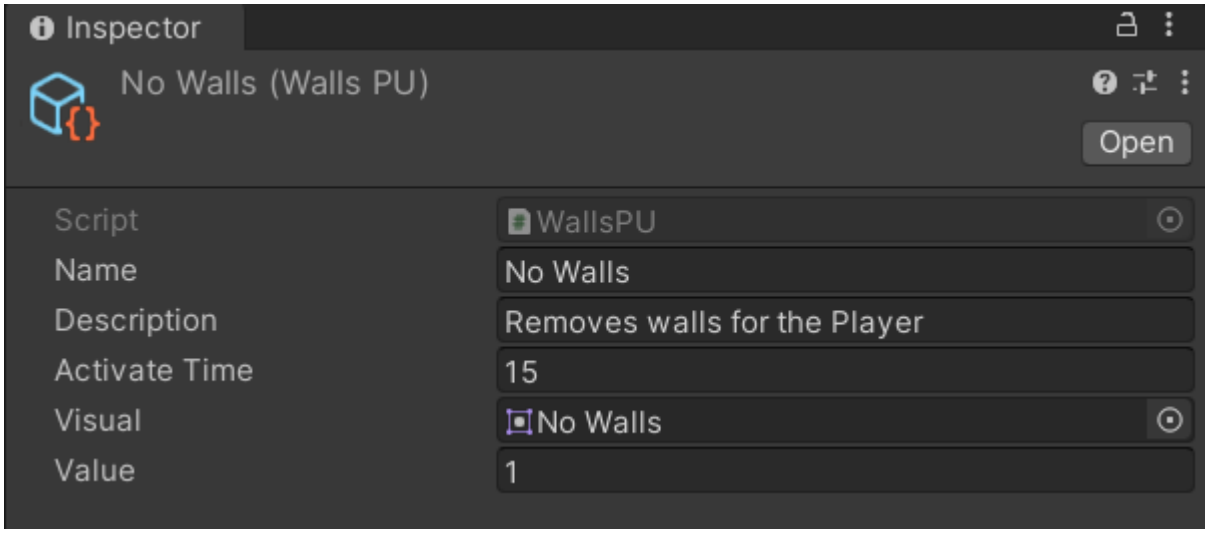


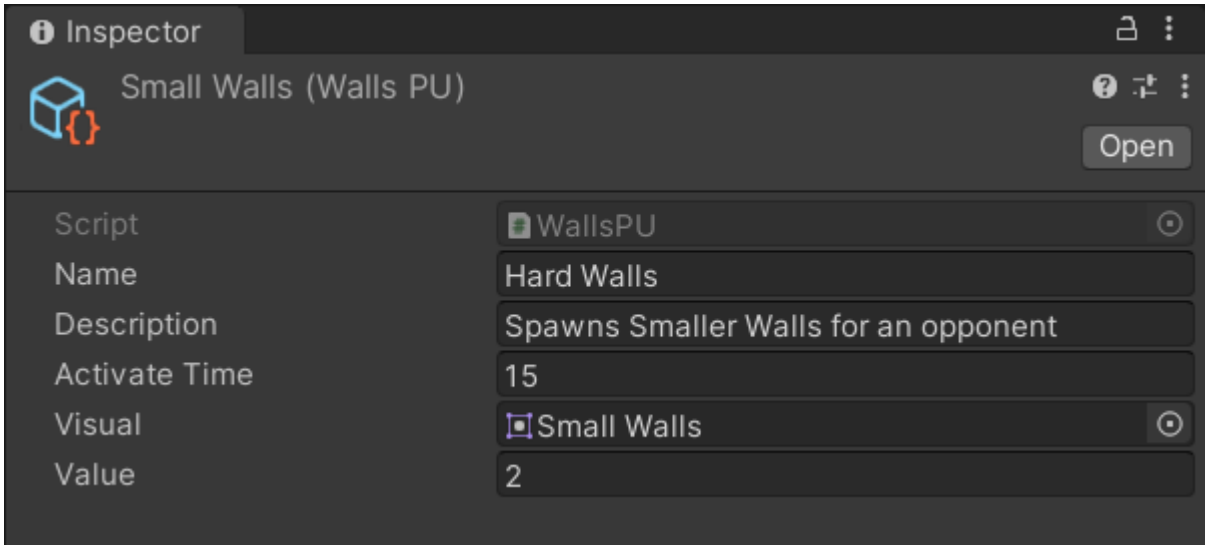

Inspector
Add 5 Length (Length PU)
Open
Script
LengthPU
Name
Add 5 Length
Description
Adds 5 segments to an Opponent
Activate Time
0
Visual
+5 Length
Value
5

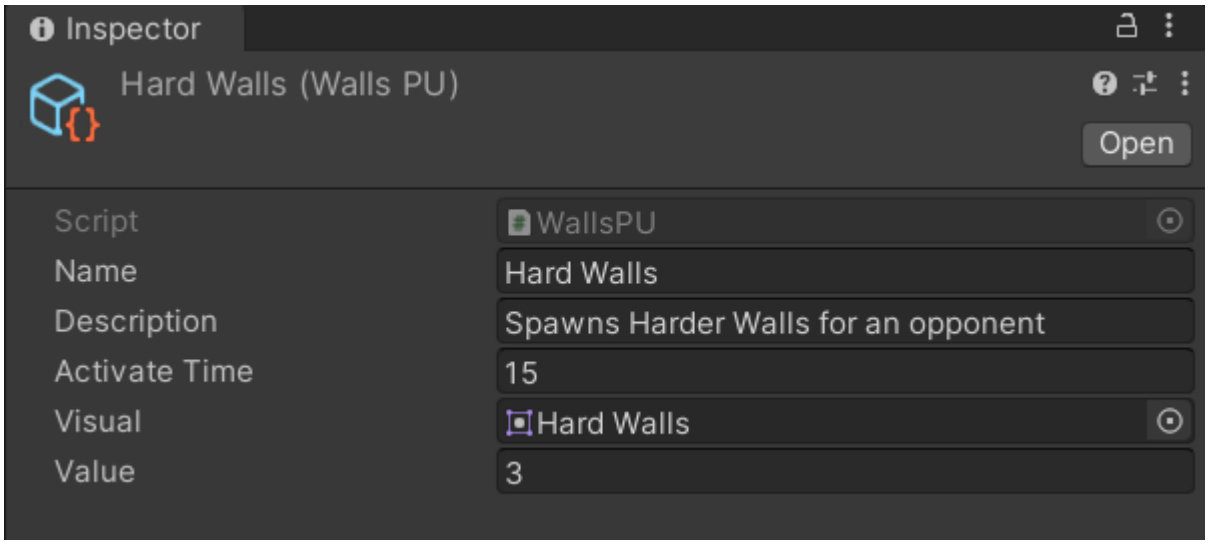

Inspector
Boost 2x (Boost PU)
Open
Script
BoostPU
Name
Boost 2x
Description
Doubles the speed of an opponent's snake for 1
Activate Time
5
Visual
2.0 speed
Value
2


Inspector
-5 Score (Score PU)
Open
Script
ScorePU
Name
-5 Score
Description
-5 an opponents FoodCount
Activate Time
0
Visual
-5 score
Value
-5

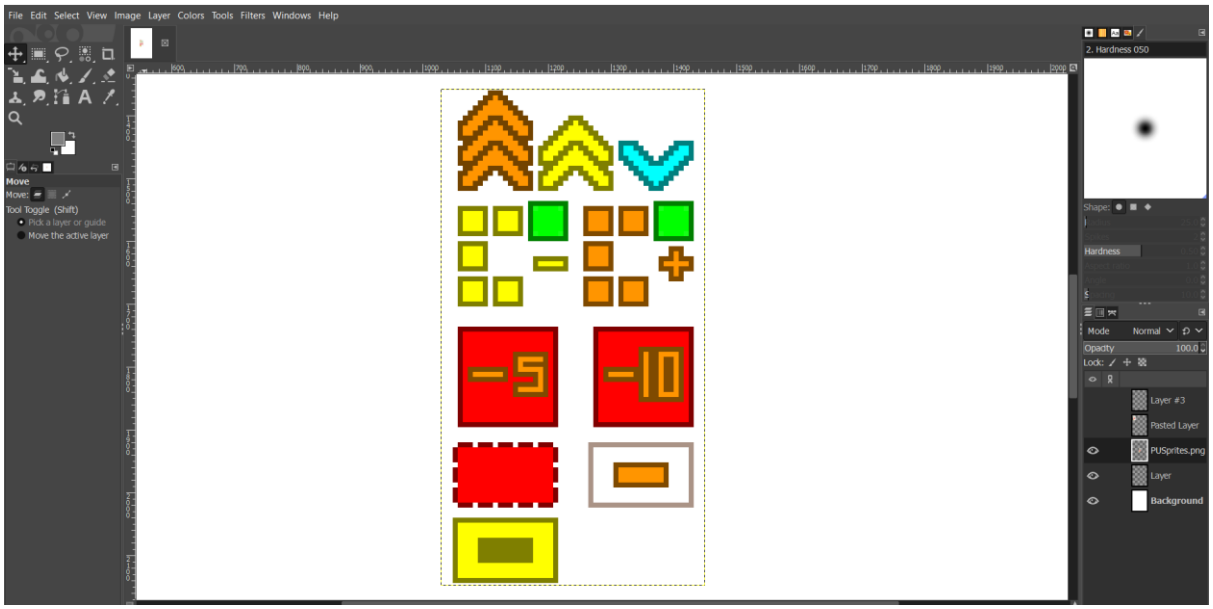

Inspector
Sub 5 Length (Length PU)
Open
Script
LengthPU
Name
Sub 5 Length
Description
Removes 5 segments from the Player
Activate Time
0
Visual
-5 Length
Value
-5


Inspector
No Walls (Walls PU)
Open
Script
WallsPU
Name
No Walls
Description
Removes walls for the Player
Activate Time
15
Visual
No Walls
Value
1

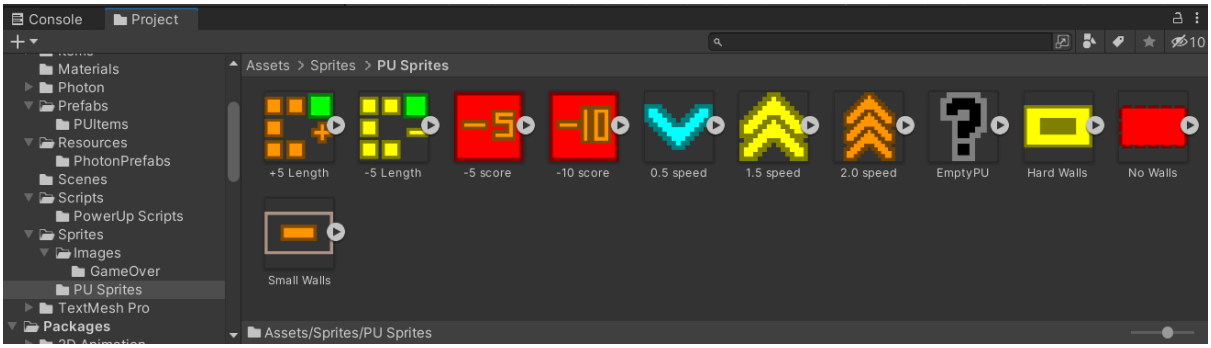

Inspector
Small Walls (Walls PU)
Open
Script
WallsPU
Name
Hard Walls
Description
Spawns Smaller Walls for an opponent
Activate Time
15
Visual
Small Walls
Value
2



Notice how these objects have images already placed in the Visual section. These sprites were all created in an image manipulation software named GIMP, resulting in a single file looking like this:



After exporting each sprite separately, they were then put within the project's sprites folder for easy access, and then assigned to their corresponding power-up object:

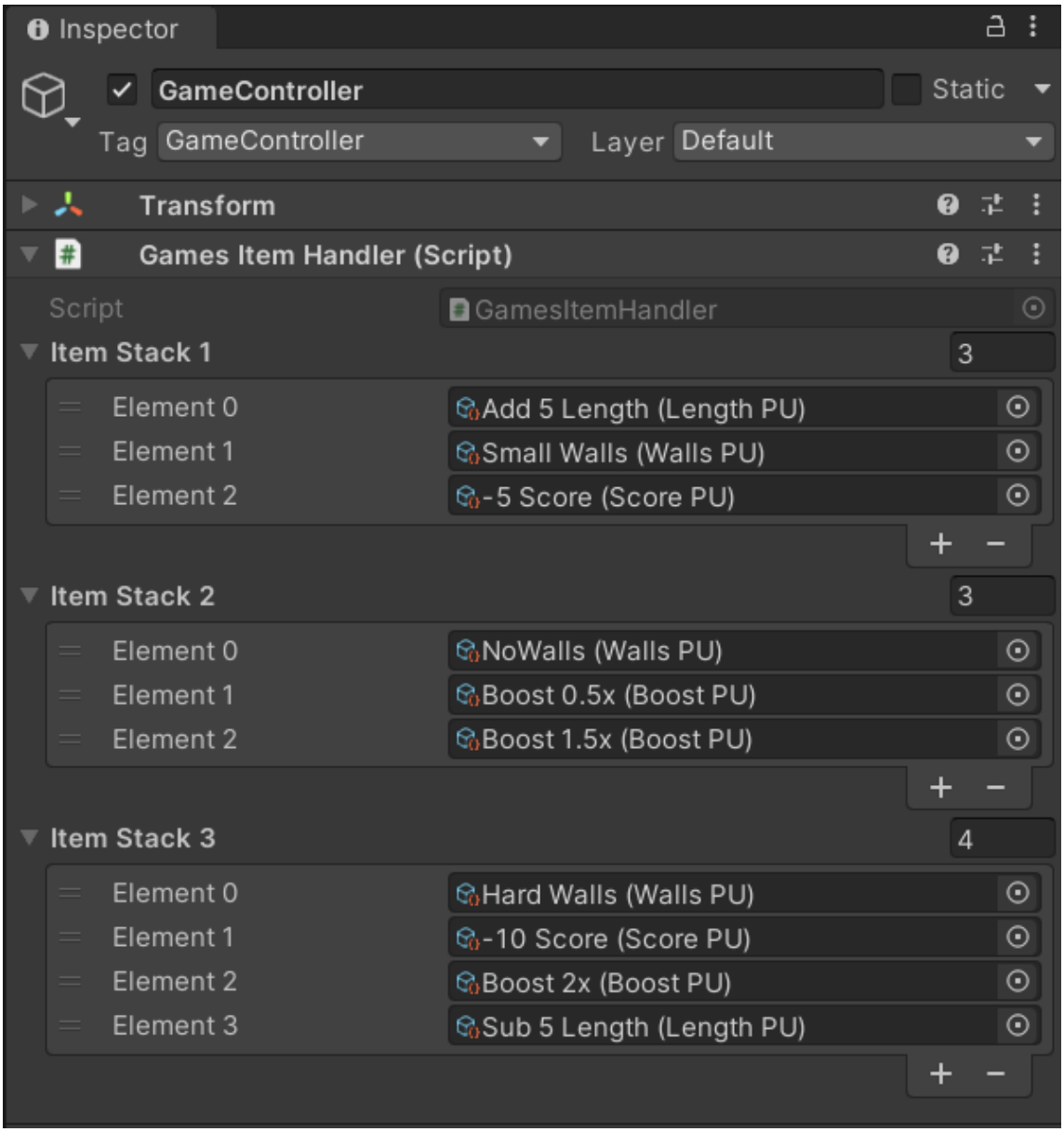


Once this is done, there has to be a way to store the power-ups as three different lists that are publicly available for all players in the room. This is done in a script named *GamedsItemHandler*, which will hold these three lists:

```
Unity Script (1 asset reference) | 2 references
public class GamesItemHandler : MonoBehaviour
{
    // Provide three different lists of power-ups for all players in the room to access
    public PUItem[] ItemStack1;
    public PUItem[] ItemStack2;
    public PUItem[] ItemStack3;
}
```

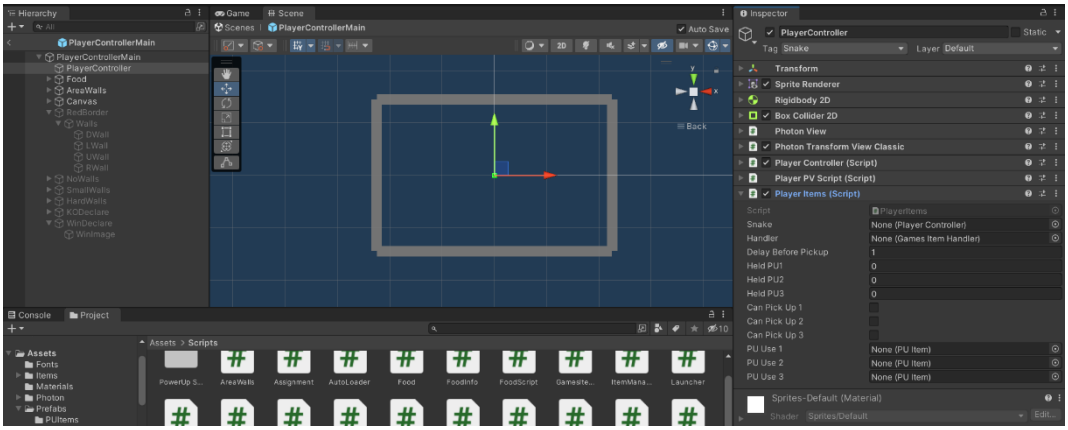
This script is placed within an empty GameObject *GameController*, not as part of the *PlayerController* prefab, but rather in the game scene itself (the same place where the *SpawnManager* is). This allows every player to access the same lists without having to have the same script on every prefab that is instantiated in a game.

This lists are then filled up with certain power-up objects as such:



Getting Power-Ups

Now, for the player themselves to get a random power-up from each *ItemStack*. A new script is made named *PlayerItems* that will manage the retrieval and usage of power-ups, and this script will be placed within the *PlayerController* prefab.



Within this code, there are references to the *PlayerController* script as well as the *GamesItemHandler* script. While the *PlayerController* variable (named Snake) can be assigned via the Unity Editor, the *GamesItemHandler* script (named *Handler*) is not as easy, as it is only seen in the game scene, and objects only in a scene cannot be referred to a prefab using the editor. Instead, when this script is activated, it will find the first *GameObject* in the Game scene with a tag of "GameController" (which is assigned to the *GameController* holding *GamesItemHandler* using the editor), and set the Handler variable to the *GamesItemHandler* script it finds within that *GameObject*:

```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    PV = GetComponent<PhotonView>(); // Grab the Photon View of the player

    if (!PV.IsMine)
    {
        // Run this script only on the local player
        return;
    }

    // Grab the GamesItemHandler script from within the Game scene
    Handler = GameObject.FindGameObjectWithTag("GameController").GetComponent<GamesItemHandler>();
```

In addition, variables holding the index of the power-up to be held from each *ItemStack*, the Power-Up object itself and a bool to check if each of the three Power-Up holders can collect power-ups are declared:

Then, three functions are called named *ResetPU1*, *ResetPU2* and *ResetPU3*:

```
// Reset the Power-Up holders, allowing them to be set active when needed
ResetPU1();
ResetPU2();
ResetPU3();
```

These functions essentially reset the variables seen above, allowing for them to be set properly when the player does first collect a power-up. The *PUUse* variable of each power-up holder is set to null to indicate a lack of object in that variable, the *heldPU* variable of each holder is set to -1, as -1

```
public int heldPU1;
public int heldPU2;
public int heldPU3;

public bool canPickUp1;
public bool canPickUp2;
public bool canPickUp3;

public PUItem PUUse1;
public PUItem PUUse2;
public PUItem PUUse3;
```

does not exist as an index in any of the three *ItemStack* lists, and *canPickUp* for each holder is set to true, to allow for a power-up to be collected in each holder (as long as requirements to pickup a power-up are met):

```
2 references
public void ResetPU1()
{
    // Reset Power-Up holder 1 related variables to allow it to pick up power-ups
    PUUse1 = null;
    heldPU1 = -1;
    canPickUp1 = true;
}

2 references
public void ResetPU2()
{
    // Reset Power-Up holder 2 related variables to allow it to pick up power-ups
    PUUse2 = null;
    heldPU2 = -1;
    canPickUp2 = true;
}

2 references
public void ResetPU3()
{
    // Reset Power-Up holder 3 related variables to allow it to pick up power-ups
    PUUse3 = null;
    heldPU3 = -1;
    canPickUp3 = true;
}
```

Whenever a player is able to pick up a certain power-up and proceeds to do so, a coroutine is called named *Pickup*, taking in an integer indicating the *Choice* of what *ItemStack* to randomise a power-up from. For example, if the player chooses to randomise a power-up from the first *ItemStack*, then the *Pickup* coroutine is started with integer 1. If its from the second *ItemStack*, the integer 2 is passed through, and so on. In each case, the *canPickUp* Boolean of the corresponding stack is set to false and the code waits for a set amount of time (currently at 1 second, will be needed for later on). Once this is done, a random number is chosen from 0 to 1 less than the length of the corresponding *ItemStack*, and the power-up at the index of that random number in that *ItemStack* is set to the *heldPU* variable. For example, if the player chooses to get a power-up from the first *ItemStack*, the following is executed:

```
1 reference
public IEnumerator Pickup(int Choice)
{
    switch (Choice)
    {
        case 1: // A power-up from ItemStack 1 is chosen
            if (heldPU1 == -1 && canPickUp1) // Only continue if the player doesn't already have a powerup from this ItemStack and is able to pick up from it
            {
                // Set canPickUp to false, so that a power-up cannot be picked again while one is already being held
                canPickUp1 = false;

                yield return new WaitForSeconds(DelayBeforePickup);

                // Choose a random power-up from ItemStack 1 to be held
                int PURand = Random.Range(0, Handler.ItemStack1.Length);
                PUUse1 = Handler.ItemStack1[PURand];
                heldPU1 = PURand;
            }
            break;
```

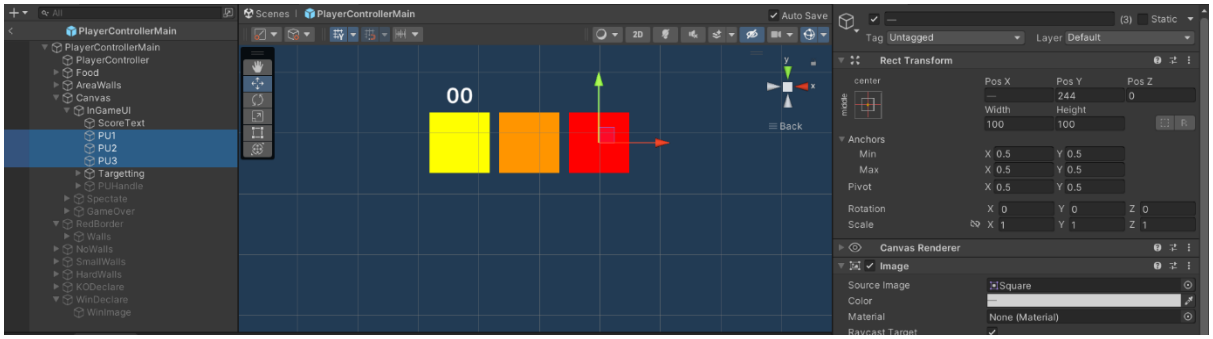
The same is repeated for cases 2 and 3, replacing every instance of “1” in this section to 2 or 3.

As this enumerator is being held in a different place from the *PlayerController* script, which manages all player inputs, a separate function has to be called to ensure that different scripts can call the

coroutine. This coroutine will be called *StartPickup* and will take in an integer between 1 and 3 as an argument, calling the *Pickup* coroutine with the same integer:

```
3 references
public void StartPickup(int Choice)
{
    StartCoroutine(Pickup(Choice));
}
```

Before moving on, we need to make UI elements for the player to indicate that a player is able to get a certain power-up. Within the *PlayerController* prefab's canvas, we can set up 3 square sprites of differing colours (to differentiate each power-up slot from each other) as such:



In the *PlayerController* script, image variables PU1, PU2 and PU3 are created, which hold the sprite objects for their corresponding power-up indicator squares:

```
[SerializeField] Image PU1;
[SerializeField] Image PU2;
[SerializeField] Image PU3;
```

Then, in the same script, a new function is called named *PUCheck*. Whenever this function is called, it checks whether the player has a score equal to or over 10, 15 or 20. If the player's score is over 10, a coroutine *PUActive* is called, passing the *PU1* variable and setting a bool named *IsPP1Active* to true. If the player's score is over 15, the same coroutine is called, passing the PU2 variable and setting a bool named *IsPP2Active* to true. If the player's score is over 20, this coroutine is called passing the *PU3* variable and setting a bool named *IsPP3Active* to true.

```
private void PUCheck()
{
    if (!PV.IsMine)
    {
        return;
    }

    if (FoodCount >= 10 && IsPP1Active == false) // If the player's score is above 9 and the player doesnt already have a power-up in slot 1:
    {
        // Get PU1 to light up and set IsPP1Active to true
        StartCoroutine("PUActive", PU1);
        IsPP1Active = true;
    }
    else if(FoodCount < 10)
    {
        // If the score is less than 10, dim PU1
        ColourReset(1);
    }
    if (FoodCount >= 15 && IsPP2Active == false) // If the player's score is above 14 and the player doesnt already have a power-up in slot 2:
    {
        // Get PU2 to light up and set IsPP2Active to true
        StartCoroutine("PUActive", PU2);
        IsPP2Active = true;
    }
    else if (FoodCount < 15)
    {
        // If the score is less than 15, dim PU2
        ColourReset(2);
    }
    if (FoodCount >= 20 && IsPP3Active == false) // If the player's score is above 19 and the player doesnt already have a power-up in slot 3:
    {
        // Get PU3 to light up and set IsPP3Active to true
        StartCoroutine("PUActive", PU3);
        IsPP3Active = true;
    }
    else if (FoodCount < 20)
    {
        // If the score is less than 20, dim PU3
        ColourReset(3);
    }
}
```

The coroutine *PUActive* essentially “brightens” the corresponding power-up indicator square, indicating it is ready to receive a power-up (it does this by having a float increment by 0.1 from 0.3 to 1, and setting the square’s colour alpha – how transparent it is - to that float in every increment), while *ColourReset* dims it, indicating it is no longer ready to receive a power-up (it does this by simply resetting the alpha of the square to 0.3):

```
0 references
IEnumerator PUActive(Image image)
{
    // Brighten the corresponding power-up indicator square over a short period of time
    for (float f = 0.3f; f <= 1; f += 0.1f)
    {
        var tempColour = image.color;
        tempColour.a = f;
        image.color = tempColour;
        yield return new WaitForSeconds(0.03f);
    }
}
```

```
8 references
private void ColourReset(int Check)
{
    switch (Check)
    {
        // Based on what power-up indicator square is set to reset, dim the alpha of that square to 0.3 and set that relative IsPPActive boolean to false
        case 1:
            var tempColour = PU1.color;
            tempColour.a = 0.3f;
            PU1.color = tempColour;
            IsPP1Active = false;
            break;
        case 2:
            tempColour = PU2.color;
            tempColour.a = 0.3f;
            PU2.color = tempColour;
            IsPP2Active = false;
            break;
        case 3:
            tempColour = PU3.color;
            tempColour.a = 0.3f;
            PU3.color = tempColour;
            IsPP3Active = false;
            break;
    }
}
```

The function *OnTriggerEnter2D* needs to be edited to account for these new functions; whenever a player collects a piece of food, *PUCheck* needs to be called, and whenever a player dies, *ColourReset* needs to be called on all the indicator squares (this also has to be done when the game starts to ensure no power-ups are active at the very beginning of a match):

```
// Calls when the Snake Head collider hits another box collider
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D other)
{
    // Do not do anything if the Snake head is not from the local player
    if (!PV.IsMine)
    {
        return;
    }

    // If the snake collides with a food GameObject:
    if (other.tag == "Food")
    {
        StopCoroutine("Movement");
        PV.RPC("RPC_Grow", RpcTarget.All);
        StartCoroutine("Movement");
        PUCheck(); // Check if a power-up is able to be collected
    }
    // Otherwise if the snake collides with either the walls or itself:
    else if ((other.tag == "Walls" || other.tag == "ObstacleS"))
    {
        while (!IsLost) // While the Player has not lost yet:
        {
            //PV.RPC("RPC_ShowKO", RpcTarget.All); // Call the RPC to show the K.O interface to all players in the room
            //StopAllCoroutines(); // Stop all Coroutines currently active
            //Die(); // Run the Die function (for the local player only)
            PV.RPC("RPC_Die", RpcTarget.All); // Run the Die RPC (for all players to see)
        }

        for (int i = 1; i < 4; i++)
        {
            ColourReset(i); // Reset all power-up indicators
        }
    }
```

```
Unity Message | 0 references
void Start()
{
    for(int i = 0; i < 4; i++)
    {
        ColourReset(i);
    }
```

In the Unity-default Update function (a function that calls once a frame), we will check if a player has pressed any of the three buttons to get a random power-up, and if so (given that they haven't lost yet and the player has enough score to get the power-up), call the *PlayerItems' StartPickup* function with the corresponding *ItemStack* list number. In addition, reduce the score of the player by either 10, 15 or 20 based on what *ItemStack* the power-up is being taken from, and reduce the length of the player's snake by the same amount (as the player is essentially "paying" score to get a power-up). Then, check if any power-up indicator squares are still going to be active after the score deduction using *PUCheck* and set the Score UI to the new player score:

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    if (!PV.IsMine || IsLost)
    {
        // Do not do anything if the player is not the local player and has lost
        return;
    }

    // If U has been pressed and the player is eligible for a power-up from ItemStack 1, "purchase" a power-up from that stack
    if (Input.GetKeyDown(KeyCode.U) && ItemsScript.heldPU1 == -1 && IsPP1Active)
    {
        ItemsScript.StartPickup(1);
        FoodCount -= 10;
        LengthChange(-10);
        PUCheck();
        ScoreText.text = FoodCount.ToString("00");
    }
```

```
// If I has been pressed and the player is eligible for a power-up from ItemStack 2, "purchase" a power-up from that stack
if (Input.GetKeyDown(KeyCode.I) && ItemsScript.heldPU2 == -1 && IsPP2Active)
{
    ItemsScript.StartPickup(2);
    FoodCount -= 15;
    LengthChange(-15);
    PUCheck();
    ScoreText.text = FoodCount.ToString("00");
}
```

```
// If O has been pressed and the player is eligible for a power-up from ItemStack 3, "purchase" a power-up from that stack
if (Input.GetKeyDown(KeyCode.O) && ItemsScript.heldPU3 == -1 && IsPP3Active)
{
    ItemsScript.StartPickup(3);
    LengthChange(-20);
    FoodCount -= 20;
    PUCheck();
    ScoreText.text = FoodCount.ToString("00");
}
```

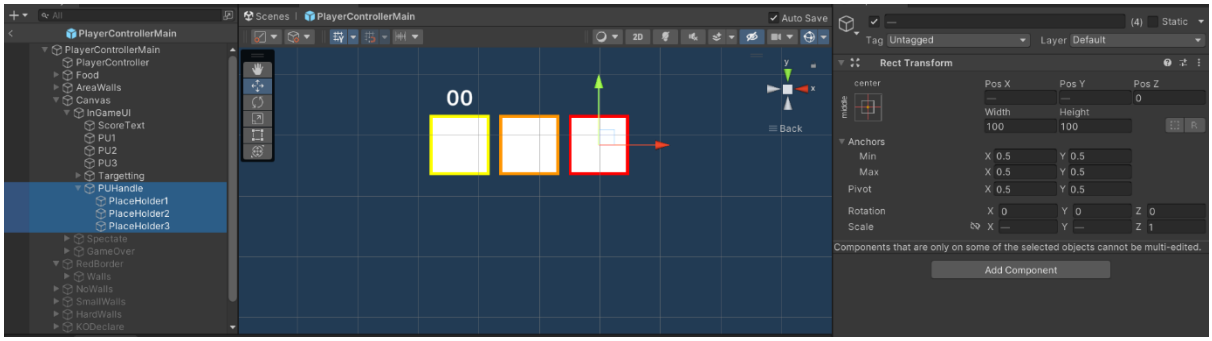
Notice that in all 3 code snippets, a function named *LengthChange* is called. This local function is responsible for either adding or removing length from a snake. To remove length from a snake (indicated by the integer value passed through the function being less than 0), the code will remove the last segment of the snake, and repeat this for the number of times stated in the value argument (after being made positive using *Mathf.Abs()*). To add length to the snake, essentially mimic what *RPC_Grow* does for the number of times as specified in the value argument. In both cases, the speed of the player needs to take into account these changes in length. To calculate the new speed, we

simply divide the default speed (time of 0.1) by 1.25 to the power of the whole number value of the player's score divided by 5:

```
public void LengthChange(int value)
{
    switch (value)
    {
        case int n when (n < 0): // When the player needs to lose length:
            for (int i = 0; i < Mathf.Abs(value); i++)
            {
                // Destroy the last segment of the player "value" number of times in the network
                int count = _segments.Count;
                GameObject s = _segments[count - 1].gameObject;
                PhotonView sPV = s.GetPhotonView();
                if (sPV.IsMine)
                {
                    _segments.Remove(s.transform);
                    PhotonNetwork.Destroy(s);
                }
                else
                {
                    continue;
                }
            }
            // Set the new player speed to the speed of the player with their new length
            repeat_time = 0.1f / Mathf.Pow(1.25f, (FoodCount / 5));
            break;
        case int n when (n > 0):
            for (int i = 0; i < value; i++)
            {
                // Instantiate the SnakeSegment for the player "value" number of times (mimicking the method in RPC_Grow)
                StartCoroutine("Wait", 0.02f);
                GameObject SnakeS = PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs", "SnakeSegment"), _segments[_segments.Count - 1].position, Quaternion.identity);
                Transform segment = SnakeS.transform;
                segment.position = _segments[_segments.Count - 1].position;
                _segments.Add(segment);
            }
            // Set the new player speed to the speed of the player with their new length
            repeat_time = 0.1f / Mathf.Pow(1.25f, (FoodCount / 5));
            break;
    }
}
```

Now that we have a way to randomly pull a power-up for the player, we still need a way to display the process of getting a power-up. To do this we make a new empty GameObject *PUHandle* within the *PlayerController* prefab's canvas. In this GameObject we implement a new script *PowerUpHandle*, which will handle the visual randomising of power-up icons until the player receives a power-up.

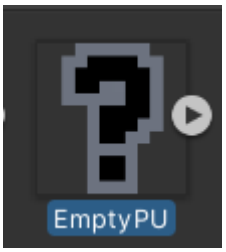
We will also put 3 white square images as children of this GameObject to act as placeholder icons:



In the *PowerUpHandle* script, slightly different functions are called based on the power-up being randomised. For example, if a power-up from the first *ItemStack* is being randomised:

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    if(SnakeItems.heldPU1 == -1)
    {
        if (SnakeItems.canPickUp1)
        {
            // As long as the player has no power-up in PU1, have the icon display in the first power-up indicator square be the EmptyItem image
            Img1.sprite = EmptyItem;
        }
        else
        {
            // If the player is starting to pick up a power-up, pick a random power-up to display
            if (shuffleSprite1)
            {
                Invoke("Shuffle1", TimeBtwShuffle); // Run the Shuffle1 function after TimeBtwShuffle seconds (set to 0.05)
                shuffleSprite1 = false; // Set shuffleSprite1 to false (means that the power-up icon won't be shuffled again until Shuffle1 sets this boolean to true again
            }
            // Repeat this until heldPU1 is no longer -1 (a power-up has been given)
        }
    }
    else
    {
        // When the player gets a power-up, set the icon as the visual sprite in the power-up object's Visual variable
        Img1.sprite = SnakeItems.PUUse1.Visual;
    }
```

This code runs in the Update function, meaning that it is called every frame. While the player does not have a power-up in slot 1 (indicated by *heldPU1* being -1), if the player has not started randomising for a power-up yet, simply set the placeholder image *Img1* to be the *EmptyItem* sprite shown below:



If the player has started randomising for slot 1, this means that the Boolean *shuffleSprite* will be true, thus allowing for the function *Shuffle1* to be invoked. Invoking a function simply means running the function after a set time delay from that like of code being executed (in this case, 0.05 seconds – this value was set in the Unity Editor). This means that right after this line, *shuffleSprite* will be set to false, stopping the two lines from running in the next frame until *Shuffle1* is called, randomising *Img1* from the list *ItemGraphics1* and setting *shuffleSprite* to true, allowing the two lines to be executed again. This is repeated until the player finally has a power-up in slot 1 (which is indicated by *heldPU1* no longer having a value of -1), in which Img1 takes on the *Visual* sprite variable set in the editor for the power-up the player now has:

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    if(SnakeItems.heldPU1 == -1)
    {
        if (SnakeItems.canPickUp1)
        {
            // As long as the player has no power-up in PU1, have the icon display in the first power-up indicator square be the EmptyItem image
            Img1.sprite = EmptyItem;
        }
        else
        {
            // If the player is starting to pick up a power-up, pick a random power-up to display
            if (shuffleSprite1)
            {
                Invoke("Shuffle1", TimeBtwShuffle); // Run the Shuffle1 function after TimeBtwShuffle seconds (set to 0.05)
                shuffleSprite1 = false; // Set shuffleSprite1 to false (means that the power-up icon won't be shuffled again until Shuffle1 sets this boolean to true again
            }
            // Repeat this until heldPU1 is no longer -1 (a power-up has been given)
        }
    }
    else
    {
        // When the player gets a power-up, set the icon as the visual sprite in the power-up object's Visual variable
        Img1.sprite = SnakeItems.PUUse1.Visual;
    }
```

```
0 references
void Shuffle1()
{
    // Randomise the icon in the first power-up indicator square
    Img1.sprite = ItemGraphics1[Random.Range(0, ItemGraphics1.Length)];
    shuffleSprite1 = true;
}
```

This makes the first power-up indicator square rapidly and randomly switch between power-up icons before it lands on the icon of the player's received power-up.

This is repeated for slots 2 and 3, allowing for all 3 indicator squares to randomise between icons when a player is trying to randomise for a power-up.

Activating a Power-Up

Now that a player is able to receive a power-up, there still needs to be a way to activate it. This will be done in the *PlayerItems* script, with functions named *ActivatePU1*, *ActivatePU2* and *ActivatePU3*.

In the example of *ActivatePU3*:

```
0 references
public void ActivatePU3()
{
    // Grab the controller and photon view of the player you are targeting
    GetController();

    if (PUUse3 is ScorePU)
    {
        // If the power-up is a score changing one, call a ScoreChange RPC to the target
        CallPV.RPC("RPC_ScoreChange", player, PUUse3.Value, PV.ViewID);
    }
    else if (PUUse3 is BoostPU)
    {
        // If the power-up is a boosting one, call a Boost RPC to the target
        CallPV.RPC("RPC_Boost", player, PUUse3.ActivateTime, PUUse3.Value, PV.ViewID);
    }
    else if (PUUse3 is LengthPU)
    {
        // If the power-up is a length changing one, call the local player's LengthChange function
        Snake.LengthChange((int)PUUse3.Value);
    }
    else if (PUUse3 is WallsPU)
    {
        // If the power-up is a wall changing one, call a WallChange RPC to the target
        CallPV.RPC("RPC_WallsChange", player, PUUse3.ActivateTime, PUUse3.Value, PV.ViewID);
    }

    // Reset the third power-up holder
    ResetPU3();
}
```

These functions act essentially the same: upon being called, they grab the Photon View and the Controller of the player that the local client is currently targeting, by calling a function named *GetController*:

```
3 references
public void GetController()
{
    // Get a list of every player in the room
    List<PhotonView> PVInstances = PlayerPVScript.GetPlayerViewList();
    foreach (PhotonView pv in PVInstances)
    {
        // If the photon view of the player being checked is the same as that of the local player's target, grab their Controller and Photon View
        if (pv.ViewID == Snake.PlayerID)
        {
            player = pv.Controller;
            CallPV = pv;
        }
    }
}
```

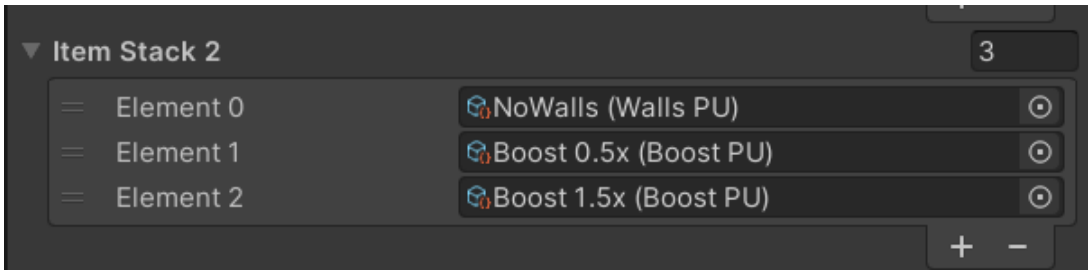
Then, they check to see what type of power-up the local player has. It does this by checking the class that power-up object is from. If the power-up is one that is executed on the local client (such as a *LengthPU* power-up in the third power-up slot), then a function/coroutine is called on that local client's *PlayerController* script (such as the *Snake*.*LengthChange* function we talked about earlier), passing through it the *ActivateTime* and Value of the power-up as arguments, if needed.

If the power-up is one that is executed on the target's client (such as a *ScorePU* power-up in the third power-up slot), then an RPC is called on that target client's *PlayerController* script (such as *RPC_ScoreChange*). In the *PlayerController* script, this RPC calls the function/coroutine needed on the target client, passing through it the *ActivateTime* and Value of the power-up as arguments, if needed, as well as the ID of the player that sent the RPC call. For example, if *RPC_ScoreChange* is called on a target within *ActivatePU1*, that target will then call the *ScoreChange* function on their own client, passing through the *ScorePU* value, and sets a integer variable *TargettedBy* to the ID

passed through the RPC as an argument. This integer value indicates the ID of the player that is targeting the local player, and is responsible for telling that player they "killed" the local player if they die with that player's ID in *TargettedBy*:

```
[PunRPC]
0 references
private void RPC_ScoreChange(float Value, int ID)
{
    if (this.PV.IsMine) // Run only on the player that this RPC is called to
    {
        // Run the player's local ScoreChange function, passing the integer passed into the RPC as an argument
        ScoreChange((int)Value);
        TargettedBy = ID;
    }
    else
    {
        return;
    }
}
```

As each *ItemStack* contains no more than one power-up from each type, the detection and execution of functions is with few lines of code. This is with the exception of the ActivatePU2 function. Here, two power-ups of class *BoostPU* are in the second *ItemStack* – one is to be called on the local player (Boost 0.5x) and the other is to be called on the target (Boost 1.5x):



To differentiate between the two power-ups, we can use a value that is different for both of them: their class value. Boost 0.5x has a value of 0.5 while Boost 1.5x has a value of 1.5. We can use this to call differing coroutines based on the value given:

```
2 references
public void ActivatePU2()
{
    // Grab the controller and photon view of the player you are targeting
    GetController();
    if (PUUse2 is BoostPU) // If the power-up is a boosting one, check the value it holds:
    {
        if (PUUse2.Value == 1.5f)
        {
            // If its value is 1.5, it is meant for the targetted player, so send RPC_Boost to the target client
            CallPV.RPC("RPC_Boost", player, PUUse2.ActivateTime, PUUse2.Value, PV.ViewID, PV.ViewID);
        }
        else
        {
            // If its value is 0.5, it is meant for the local player, so call the Boost coroutine on the local client
            StartCoroutine(Snake.Boost(PUUse2.ActivateTime, PUUse2.Value));
        }
    }
    else if (PUUse2 is WallsPU)
    {
        StartCoroutine(Snake.WallsChange(PUUse2.ActivateTime, PUUse2.Value));
    }

    ResetPU2();
}
```

Back in the *PlayerItems* script, the last part of *ActivatePU3* consists of resetting the third power-up holder with *ResetPU3* to stop the player activating the same power-up again until they can "purchase" another one.

This process is repeated for functions *ActivatePU1* and *ActivatePU2*, adding their own checks based on what power-ups are in their respective *ItemStacks*.

Power-Up functions

Now the power-ups themselves need to be coded in, which will be done in the *PlayerController* script. The RPC functions for non-local players have already been explained, as all they do is run the corresponding local functions for the targeted player.

Boosting

Boosting a player will be done as a coroutine named *Boost*, as the player will only be boosted for a set amount of time (that being 10 seconds). Within this coroutine, the *repeat_time* variable of the player is divided by the integer *mult* passed into the function. This means that, if the player needs to slow down, the value of *mult* passed into *Boost* needs to be lower than 1, as dividing *repeat_time* by a value lower than 1 increases the time spent waiting between moving the snake 1 unit. The opposite applies for speeding up the snake by passing a value of *mult* higher than 1 to reduce *repeat_time*.

The code then waits for 10 seconds before reverting the *repeat_time* to what it should be at the player's snake's current length:

```
2 references
public IEnumerator Boost(float time, float mult)
{
    // Divide the current repeat_time by the multiplier passed into the function and wait 10 seconds before reverting repeat_time to normal
    repeat_time = (0.1f / Mathf.Pow(1.25f, (FoodCount / 5))) / mult;
    yield return new WaitForSeconds(time);
    repeat_time = 0.1f / Mathf.Pow(1.25f, (FoodCount / 5));
}
```

Changing a player's length

Changing the player's snake length has already been explained with the *LengthChange* function, which either adds to or removes from the snake a set number of segments (which will be the Value integer in the power-up object that the player activates). As there is no form of waiting needed here, this is not coded in as a coroutine.

Here is the function again for reference:

```
5 references
public void LengthChange(int value)
{
    switch (value)
    {
        case int n when (n < 0): // When the player needs to lose length:
            for (int i = 0; i < Mathf.Abs(value); i++)
            {
                // Destroy the last segment of the player "value" number of times in the network
                int count = _segments.Count;
                GameObject s = _segments[count - 1].gameObject;
                PhotonView sPV = s.GetPhotonView();
                if (sPV.IsMine)
                {
                    _segments.Remove(s.transform);
                    PhotonNetwork.Destroy(s);
                }
                else
                {
                    continue;
                }
            }
            // Set the new player speed to the speed of the player with their new length
            repeat_time = 0.1f / Mathf.Pow(1.25f, (FoodCount / 5));
            break;
        case int n when (n > 0):
            for (int i = 0; i < value; i++)
            {
                // Instantiate the SnakeSegment for the player "value" number of times (mimicking the method in RPC_Grow)
                StartCoroutine("Wait", 0.02f);
                GameObject SnakeS = PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs", "SnakeSegment"), _segments[_segments.Count - 1].position, Quaternion.identity);
                Transform segment = SnakeS.transform;
                segment.position = _segments[_segments.Count - 1].position;
                _segments.Add(segment);
            }
            // Set the new player speed to the speed of the player with their new length
            repeat_time = 0.1f / Mathf.Pow(1.25f, (FoodCount / 5));
            break;
    }
}
```

Changing a player's score

Removing from a player's *FoodCount* is done with a function named *ScoreChange*, taking in an integer value for how much to change the *FoodCount* by.

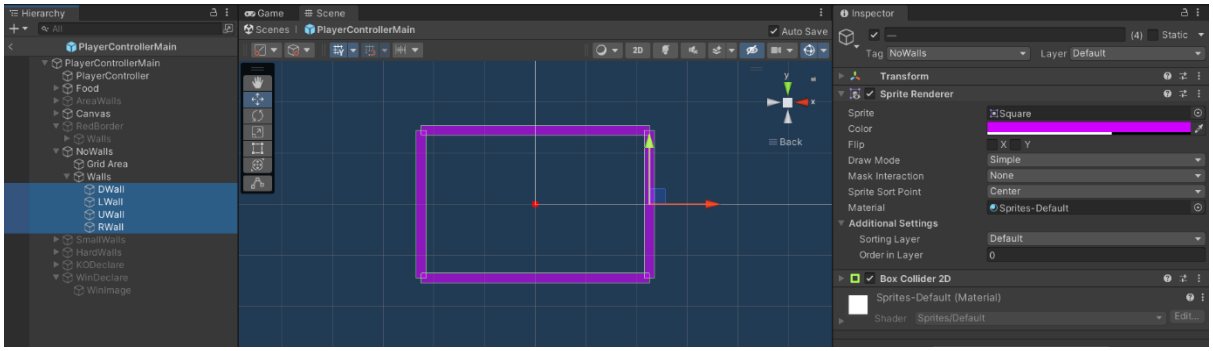
```
1 reference
public void ScoreChange(int value)
{
    // Add the integer passed through to the player's FoodCount
    FoodCount += value;

    // Update the player's Score UI to reflect these changes
    ScoreText.text = FoodCount.ToString("00");

    // Check if the player is still able to "purchase" power-ups or not
    PUCheck();
}
```

This function first adds the value passed through to the player's current *FoodCount*, and then updates the Score UI element of that player to ensure it still displays the correct score. Finally, the code calls *PUCheck* to see if the player who had their score changed is still able to "purchase" any power-ups or not.

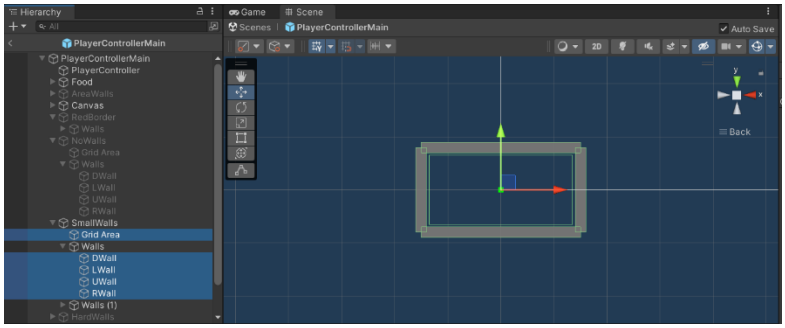
Changing a player's walls

Changing what walls a player has is more complex than the other 3 functions. First off, the wall themselves need to be changed/added.

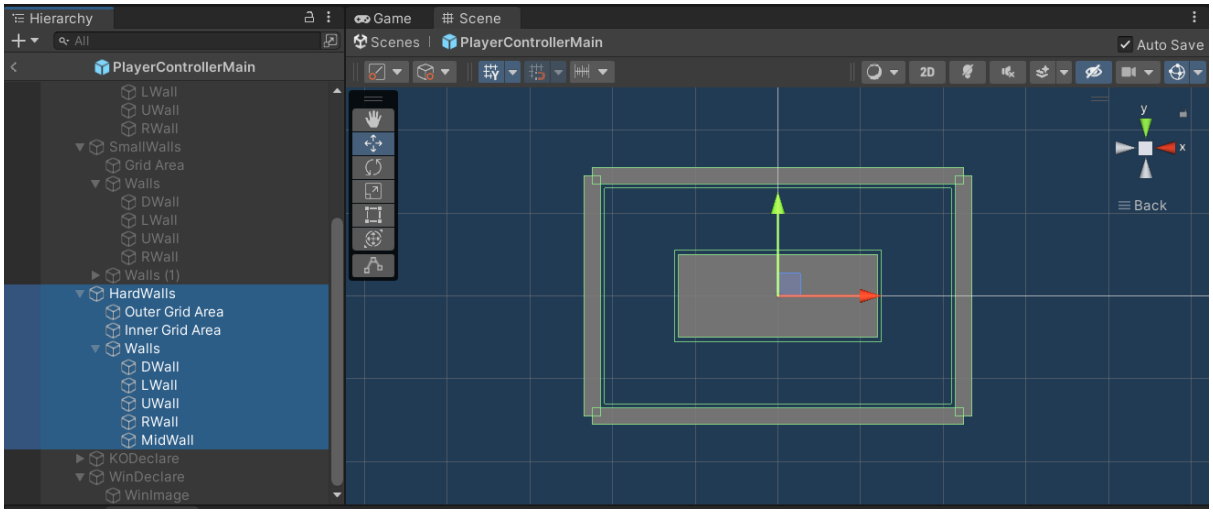
First off, a set of walls need to be added that do not kill the player upon collision, but instead moves the snake head to the other side of the player's play area, giving the illusion of a smooth "teleportation" effect. These walls will look the same as the *AreaWalls* GameObject, but with the colour of the walls being a translucent purple instead of a gray, and with the walls being tagged with "NoWalls":



Then, a set of walls need to be added that will look the same as the *AreaWalls* GameObject, but are smaller (and have a smaller *GridArea*). These walls will be tagged the same as the walls in *AreaWalls*:

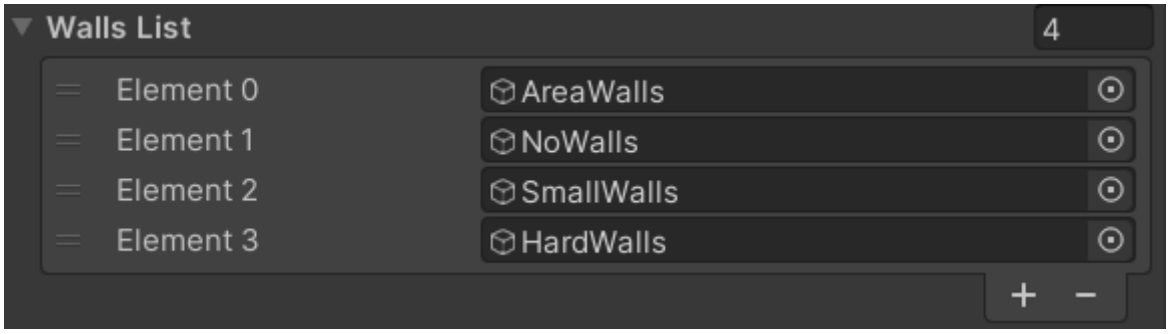


Finally, a set of walls needs to be made with an extra section in the middle to make them harder to navigate in. An extra Grid Area needs to be made to account for this added segment:



Next, the single variable holding the *AreaWalls* needs to change into a list holding all 4 kinds of walls that will be used here:

```
public List<GameObject> WallsList;
```



Next, in the *FoodScript*, the method of getting the walls and *GridArea* of the player's active walls at any time needs to be changed. A new function named *SetWallsObj* is created within this script:

```
4 references
public void SetWallsObj()
{
    // Check each walls GameObject in the list of walls to get the one that is active in the game scene
    foreach(GameObject wall in WallsList)
    {
        if (wall.activeSelf)
        {
            AreaWallsObj = wall;
        }
    }
    // Assign the gridArea of the walls to the first BoxCollider2D the code finds in the children of the walls GameObject
    gridArea = AreaWallsObj.GetComponentsInChildren<BoxCollider2D>()[0];

    // If the walls GameObject has more than 1 BoxCollider2D component in its children, assign innerArea to the second BoxCollider2D component
    if(AreaWallsObj.GetComponentsInChildren<BoxCollider2D>().Length  == 2)
    {
        innerArea = AreaWallsObj.GetComponentsInChildren<BoxCollider2D>()[1];
        HasInner = true; // Notify the code that the player has inner walls currently active
    }
    else
    {
        HasInner = false;
    }
}
```

First, this function sets the GameObject variable *AreaWallsObject* defined beforehand to the set of walls that is currently active in the list of wall GameObjects by incrementing through each wall GameObject and checking if it is currently active in the scene. Once it finds the active walls, it sets the BoxCollider2D variable *gridArea* - also defined beforehand - to the first *BoxCollider2D* component it finds within the children of *AreaWallsObj*.

However, walls such as *HardWalls* will have 2 Grid Areas, and thus will need some way to differentiate between the two. To do this, the code checks to see whether or not the number of *BoxCollider2D* components it finds within the children of *AreaWallsObj* is 2. If it is, that means the *HardWalls* is currently active, and thus a new BoxCollider2D variable *innerArea* is set to the second BoxCollider2D component found within *AreaWallsObj*. Boolean variable *HasInner* is also set to true to notify other functions that the player has 2 grid areas.

The *RandomPos* function of the food must also be edited to account for these changes. Only one line is added, checking that if the player has *HardWalls* (indicated by *HasInner* being true) and that the food's randomised position is within the inner wall of *HardWall*, randomise the position again, and keep doing so until the food is no longer within the inner wall:

```
4 references
public void RandomPos()
{
    // Grab the bounds of the Grid Area and get random x and y values from these bounds
    Bounds bounds = gridArea.bounds;
    float x = Random.Range(bounds.min.x, bounds.max.x);
    float y = Random.Range(bounds.min.y, bounds.max.y);

    // Set the positon of the food to the x and y values obtained
    this.transform.position = new Vector3(Mathf.Round(x), Mathf.Round(y), 0.0f);

    // Keep randomising the food's position until it is no longer inside the middle wall in HardWalls, should it be active
    while (HasInner == true && innerArea.bounds.Contains(this.transform.position)) {this.transform.position = new Vector3(Mathf.Round(x), Mathf.Round(y), 0.0f);}
}
```

The *SetWallsObj* and *RandomPos* functions must be called at the start of the game as such:

```
Unity Message | 0 references
void Awake()
{
    SetWallsObj();
    PV = GetComponent<PhotonView>();
    RandomPos();
}
```

Back in the *PlayerController* script, when the *WallsChange* coroutine starts, the default *AreaWalls* of the player is disabled and the walls at the index of the walls list provided by the float variable type passed into the coroutine (turned into an integer value to allow an index to be formed of it). Then, the functions *SetWallsObj* and *RandomPos* from the *FoodScript* are called to allow the food to recognise its new surroundings, before the code waits for a certain amount of time. Once this time finishes, the walls are reverted back to the default *AreaWalls* GameObject and the food gets *SetWallsObj* and *RandomPos* called again:

```
2 references
public IEnumerator WallsChange(float time, float type)
{
    // Set the active walls to be the ones at the provided index of the walls list
    WallsList[0].SetActive(false);
    WallsList[(int)type].SetActive(true);
    // Allow the food GameObject to recognise its new surroundings
    foodScript.SetWallsObj();
    foodScript.RandomPos();

    // Wait a set period of time before reverting the active walls to the default AreaWalls and allowing the food to recognise its surroundings again
    yield return new WaitForSeconds(time);

    WallsList[(int)type].SetActive(false);
    WallsList[0].SetActive(true);
    foodScript.SetWallsObj();
    foodScript.RandomPos();
}
```

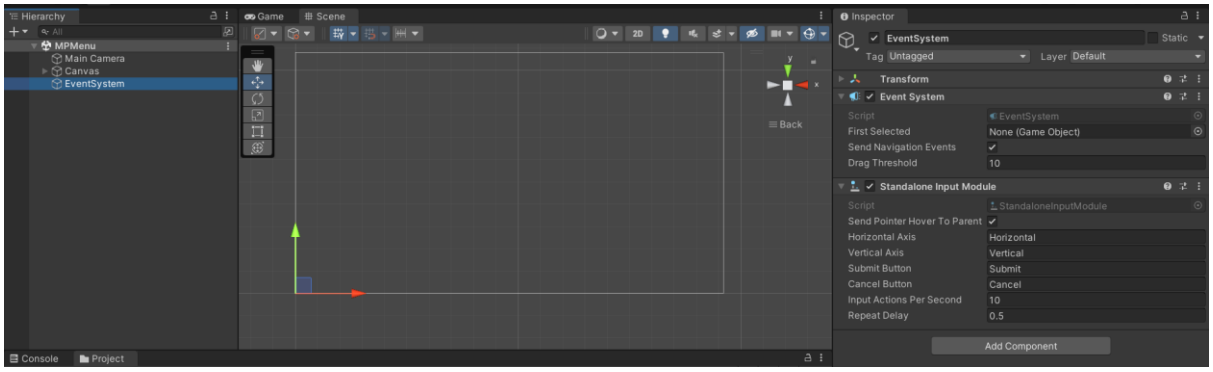
Spectating and Leaving

This section will be relatively short and will focus on the two buttons a player sees when they lose or win a game: Spectate, and Exit.

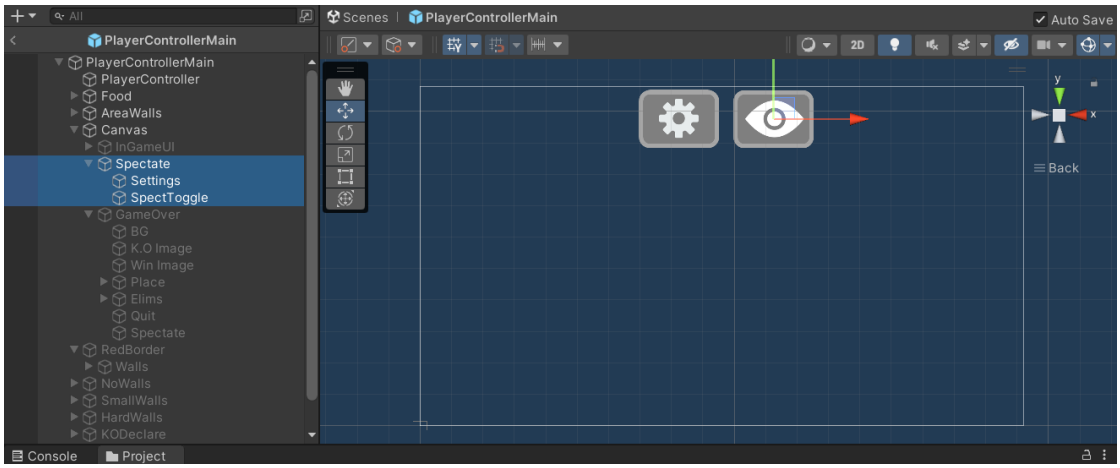
Spectating

When a player clicks the Spectate button, they should be able to see the game in action. This means disabling the Game Over UI and activating a new set of UI elements that allow the player to better see the game as a whole.

Before anything is done, the Game scene must have an empty GameObject with an *EventSystem* component added to it to allow for buttons to be clicked and for them to trigger functions:



The two spectating UI elements are buttons: a button to return to the Game Over menu and a Settings menu button.

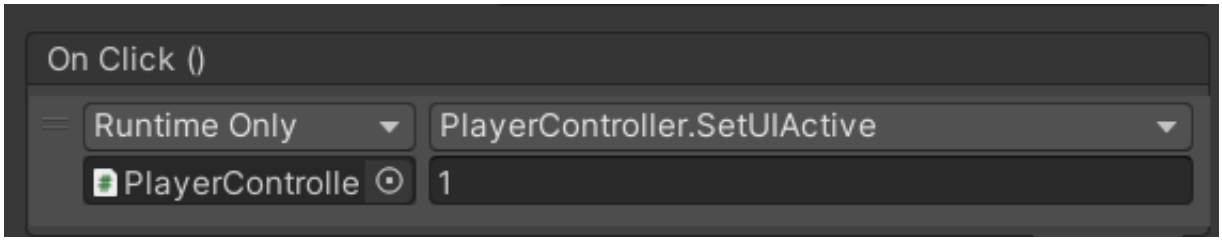


Using the *OnClick* functionality that the Unity Editor provides, we can assign functions to call when the button is clicked.

On the Spectate button within the Game Over UI, the following function is called:

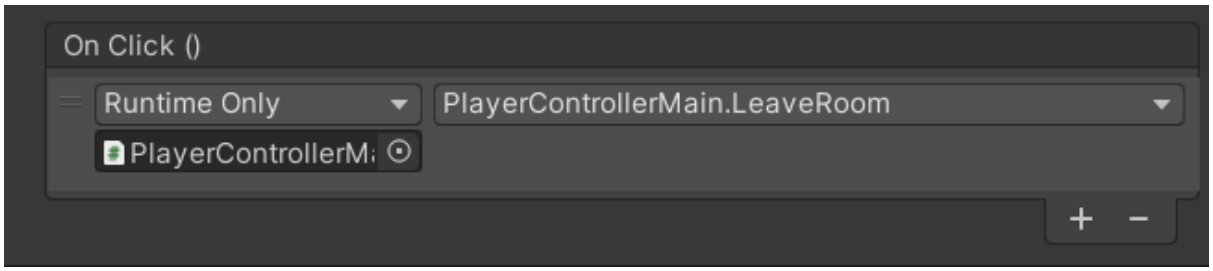
This function *SetUIActive* has already been explained, but in short, this function disables all UI in the canvas and only enables the UI GameObject at the index provided (in this case 2 – the 3rd UI GameObject for spectating is enabled).

When in the Spectate menu, the button with the eye icon has the same function called upon pressing, only opening up the Game Over UI instead of the Spectate menu:



Quitting

Quitting the game is done only when the player clicks the "Quit" button at the Game Over UI, calling the following:



This function exists on the *PlayerControllerMain* script, and will call a coroutine named *LeaveRoomEnum*:

```
// Calls when a player leaves a room (for the player only) (Button Trigger)
public void LeaveRoom()
{
    StartCoroutine(LeaveRoomEnum());
}

public IEnumerator LeaveRoomEnum()
{
    // Allow this player to switch scenes without forcing all other players to switch scenes
    PhotonNetwork.AutomaticallySyncScene = false;
    PhotonNetwork.LeaveRoom(); // Leave the Photon Room
    while (PhotonNetwork.InRoom)
        yield return null; // Do not do anything while the player is still trying to leave the room
    SceneManager.LoadScene("MPMenu"); // Load the Multiplayer Menu when the player finishes leaving
}
```

This coroutine allows for the user to switch scenes without forcing other players in the game to do the same by setting *AutomaticallySyncScene* to false. After doing so, the code makes the player leave the room (with a Photon provided function *LeaveRoom*), not allowing the player to do anything while they are trying to leave the room. Upon the player having fully left the room, the Multiplayer Menu scene can finally be opened:

Testing current project

Testing spectating went as follows:

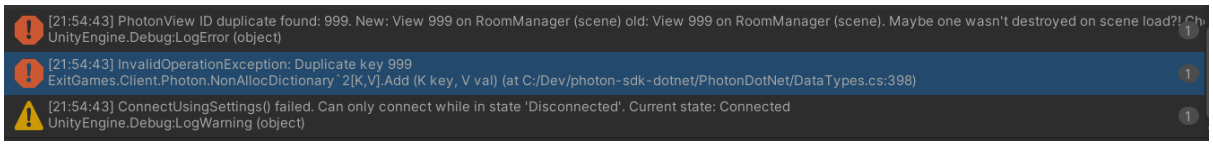
Test	Expected Output	Actual Output
Pressing the Spectate Button in the Game Over UI	Spectate UI opens	*PASS*
Pressing the Eye Button in the Spectate UI	Game Over UI opens	*PASS*

A video testing this is below:

Testing Quitting went as such:

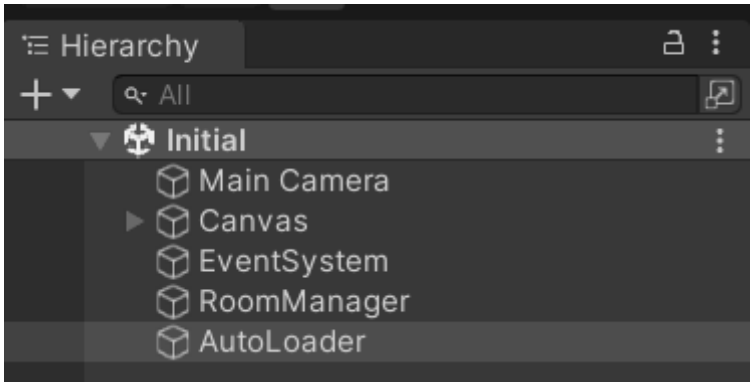
Test	Expected Output	Actual Output
Pressing the Quit Button in the Game Over UI	Multiplayer Menu scene opens	*PASS*¸ but two errors are thrown
Creating/Joining a room once leaving an old one	New room is made, all players join	Players sent to an empty game scene

The errors that occur from these tests are:



These occur due to there being multiple instances of the *RoomManager* for each player as soon as the player leaves a room. This is because whenever the player enters the Multiplayer Menu scene, the *RoomManager* is present, and as it has *DontDestroyOnLoad*, it will continue to exist even when the scene it is in is inactive. Thus, when a player enters the scene again, another instance of the *RoomManager* with ID 999 is made, causing a conflict.

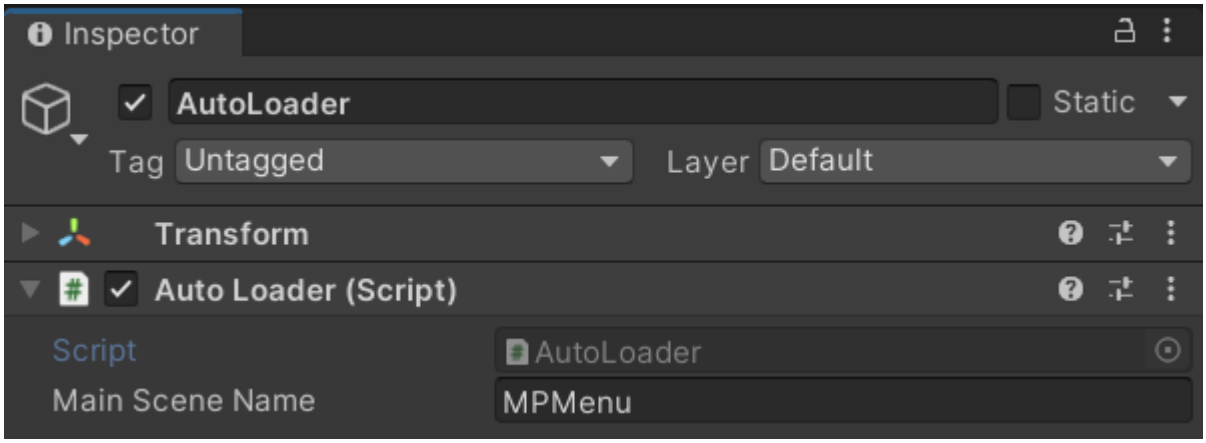
Solving this issue consists of creating a new scene that the player first loads to, before it auto-loads to the Multiplayer Menu scene. This scene will hold the *RoomManager*, so there is only one instance of it per player as it only loads once:



The *AutoLoader* GameObject holds a script also named *AutoLoader* that automatically loads the Multiplayer Menu scene, and then sets itself to inactive:

```
Unity Script (1 asset reference) | 0 references
public class AutoLoader : MonoBehaviour
{
    public static bool initialSceneLoaded;
    [SerializeField] string mainSceneName;

    Unity Message | 0 references
    private void Update() // Runs every frame
    {
        if (initialSceneLoaded)
        {
            // In the case the Initial scene is already loaded, do not run any code
            Debug.LogWarning("The Initial scene has already been loaded. This is not allowed. Cancelling main menu auto-load.");
            return;
        }
        initialSceneLoaded = true; // If Initial is not loaded, set the Loaded boolean to true

        //
        SceneManager.LoadScene(mainSceneName);
        gameObject.SetActive(false);
    }
}
```

In the Editor, we set *mainSceneName* to the name of the Multiplayer Menu scene:



We also edit the *LeaveRoom* function in the *PlayerControllerMain* script like so:

```
// Calls when a player leaves a room (for the player only) (Button Trigger)
0 references
public void LeaveRoom()
{
    PhotonNetwork.AutomaticallySyncScene = false;
    PhotonNetwork.LeaveRoom(); // Notify the server that the player has left
    SceneManager.LoadScene("MPMenu");
}
```

All that is changed is that the while loop checking if a player is still leaving a room, as that does not change the overall end result, which is the player joining the Multiplayer Menu scene.

Now, testing Quitting went as follows:

Test	**Expected Output**	**Actual Output**
Pressing the Quit Button in the Game Over UI	Multiplayer Menu scene opens	*PASS*
Creating/Joining a room once leaving an old one	New room is made, all players join	*PASS*

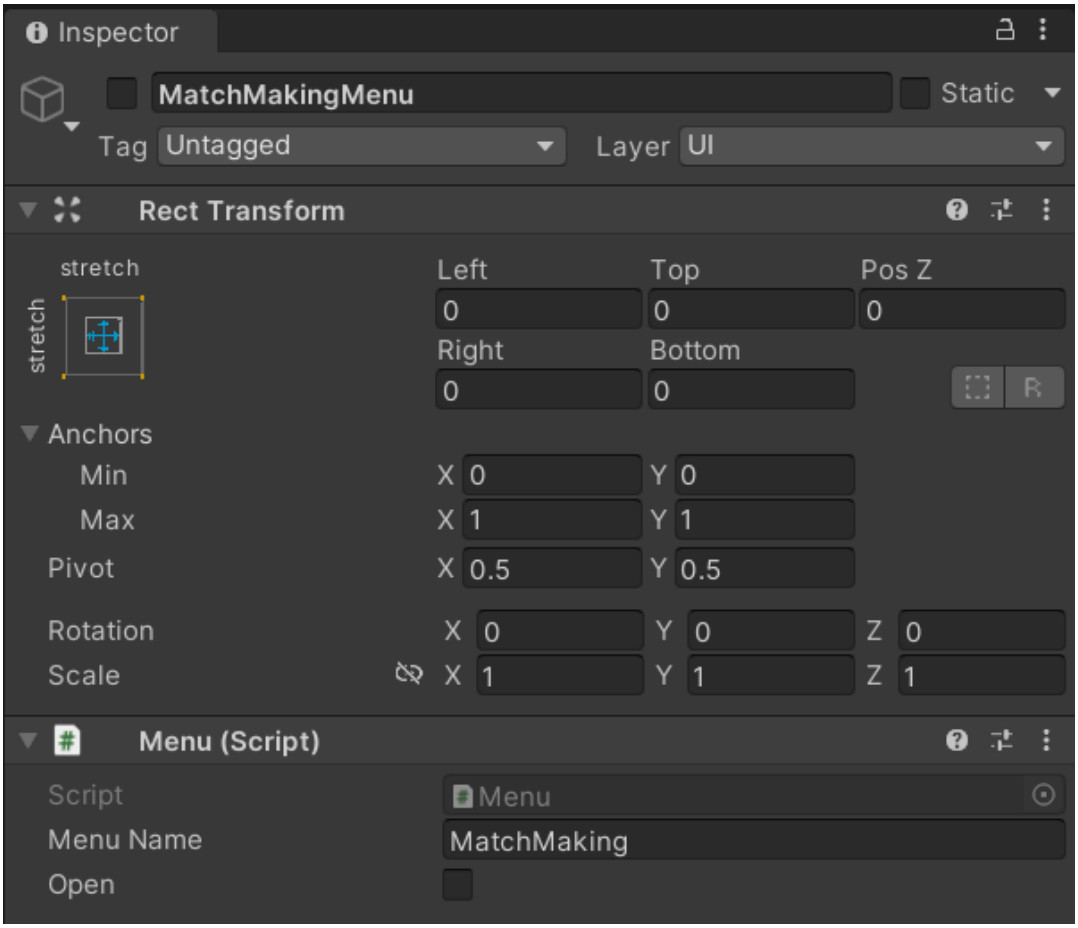
A video showing this testing is below:

Menus and Settings

This section is dedicated to the making of a Title and Settings menu, as well as changing how some menus work, and some UI elements to look more aesthetically pleasing to the player.

Adding a Title Menu

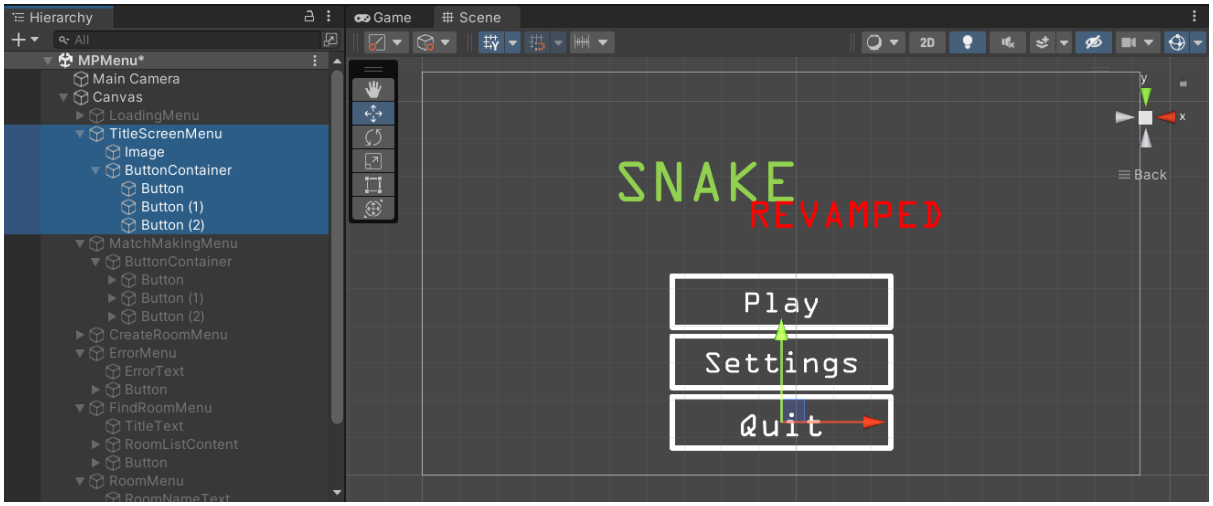
First, to differentiate the Multiplayer Matchmaking menu from the Title menu within the *MPMenu* scene, I will rename *TitleMenu* to *MatchMaking* Menu, renaming it in the *Menu* script too:



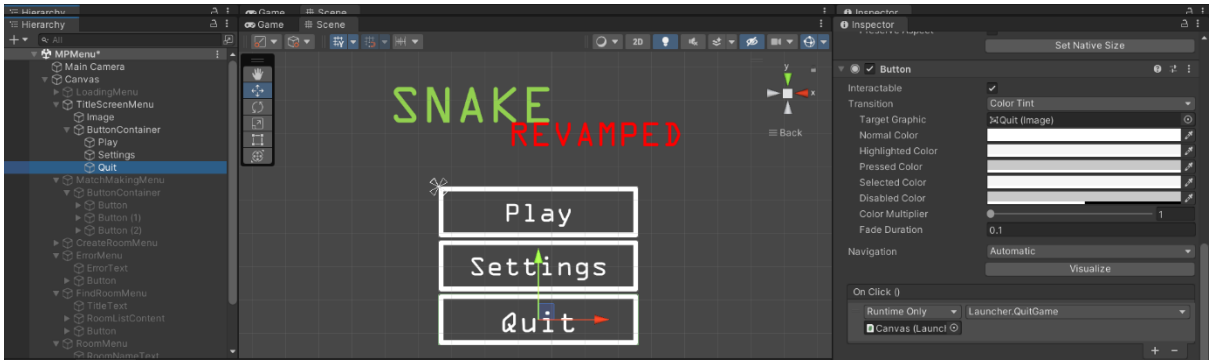
Then, any references to opening the "Title" Menu when leaving a room in the Launcher script will now be changed to open the "MatchMaking" menu:

```
// Calls when a player has finished leaving a room (local player only)
4 references
public override void OnLeftRoom()
{
    // Send the player to the Title Menu
    MenuManager.Instance.OpenMenu("MatchMaking");
}
```

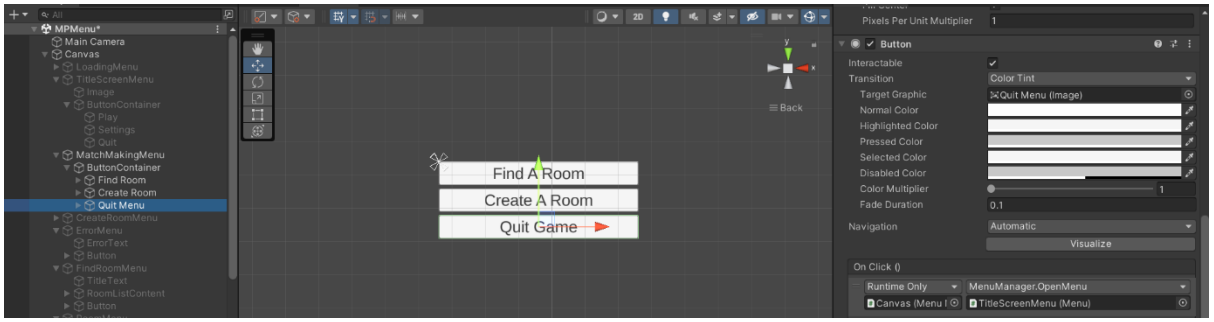
A new menu in the *MPMenu* scene is created named *TitleScreenMenu*. This menu will consist of the title of the game, as well as three buttons allowing the player to enter the *MatchMaking* and Settings menus, as well as quit the game:



The play button has its *OnClick* function set to opening the *MatchMaking* menu, and the quit button will have its *OnClick* function set to the Launcher's *QuitGame* function. The settings button will be worked on later on:



This means that, in the *MatchMaking* menu, the Quit Game button needs to be reassigned to only open the Title Screen menu:

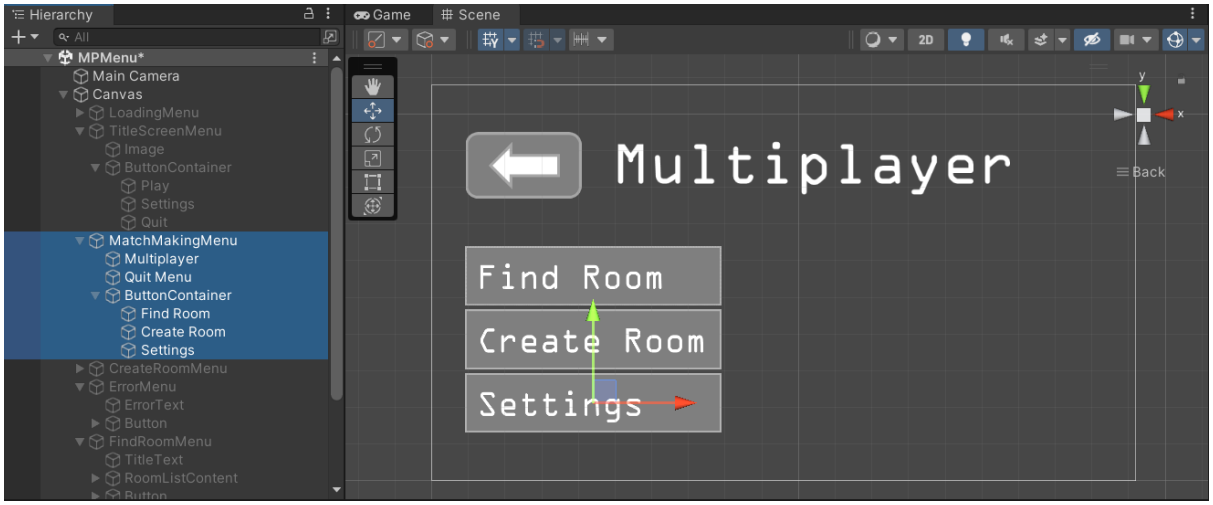


When a player joins the Master Server and then a lobby, they must be redirected to the title screen menu, so *OnJoinedLobby* must be edited as such:

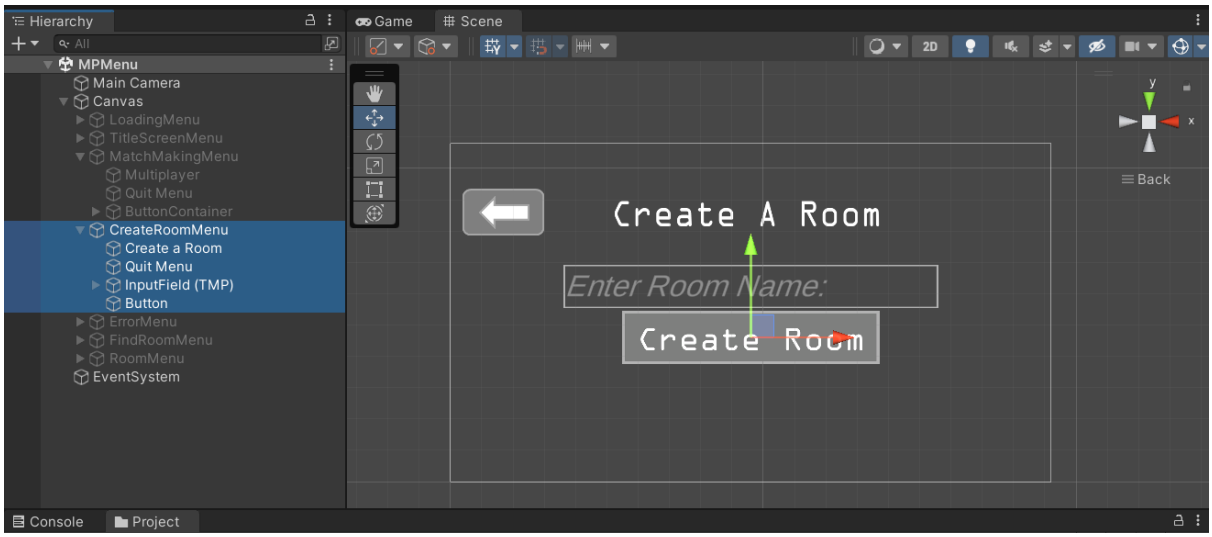
```
// Calls upon joining a Lobby
3 references
public override void OnJoinedLobby()
{
    // Opens the Title menu
    MenuManager.Instance.OpenMenu("MatchMaking");
    Debug.Log("Joined the Lobby");

    // Assign each player a random number
    PhotonNetwork.NickName = "Player " + Random.Range(0, 1000).ToString("0000");
}
```

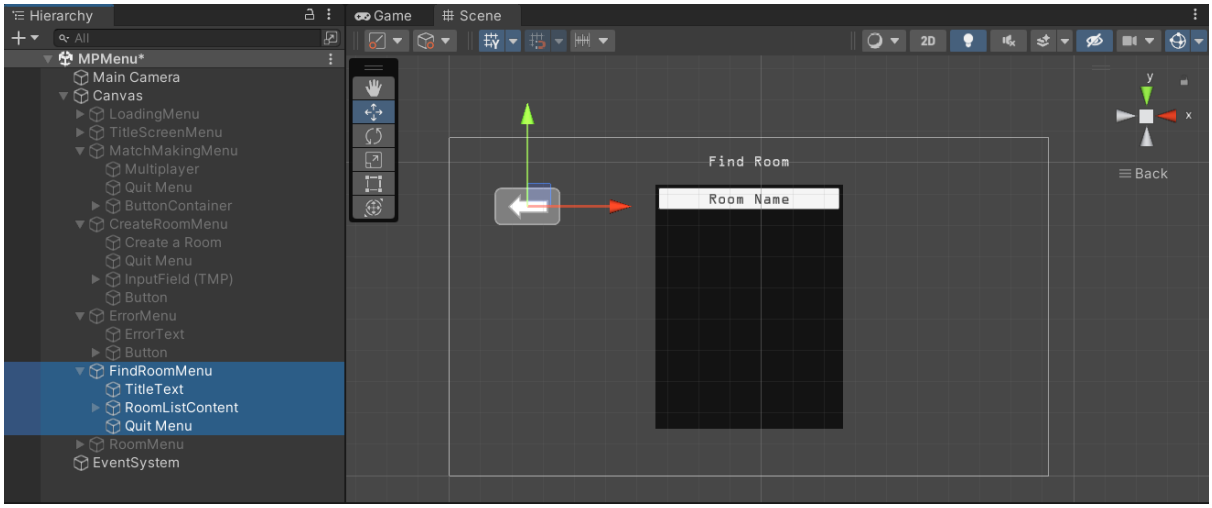
Editing Existing Menus

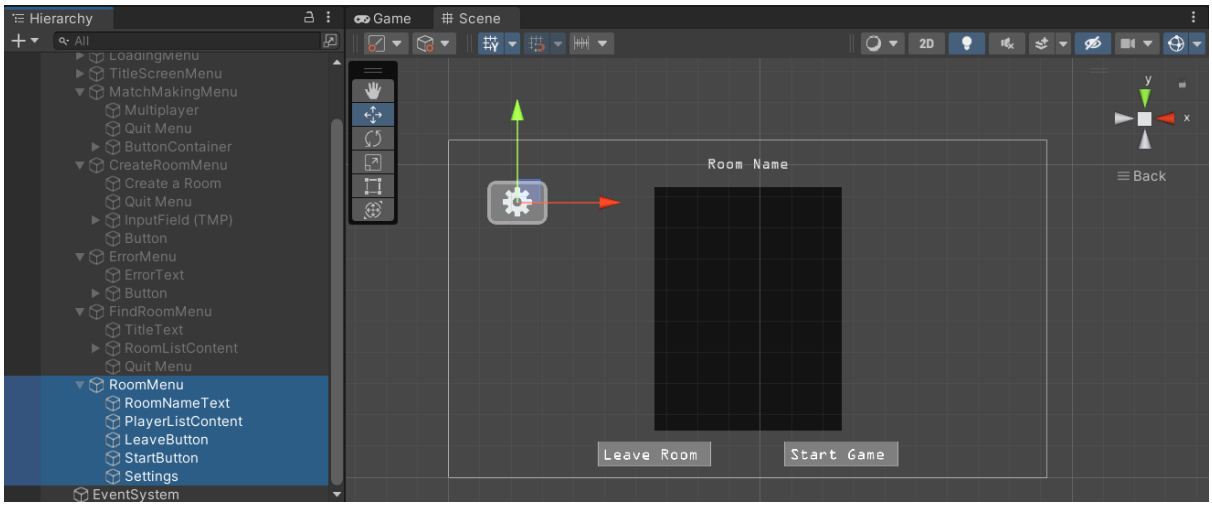
While in the MatchMaking menu, we will also change the look of the buttons and add an indicator telling the player they are in the MatchMaking menu. A button to access the settings menu will be added. We will finally change the positions of these buttons to make them more aesthetically pleasing:



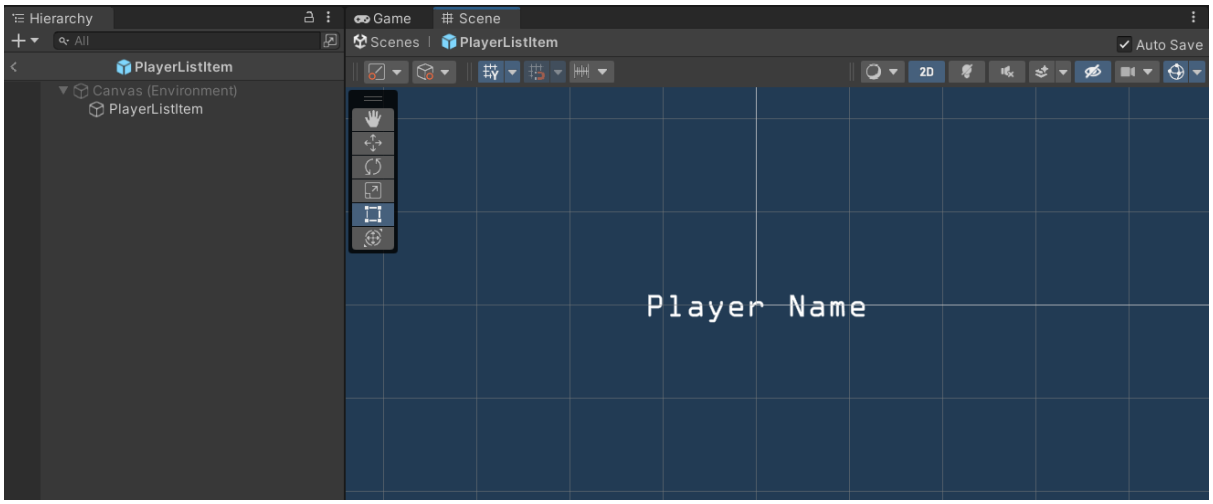
The Create Room, Find Room and menus are also readjusted and fitted with new icons to make them look nicer, and any settings buttons are added where necessary:

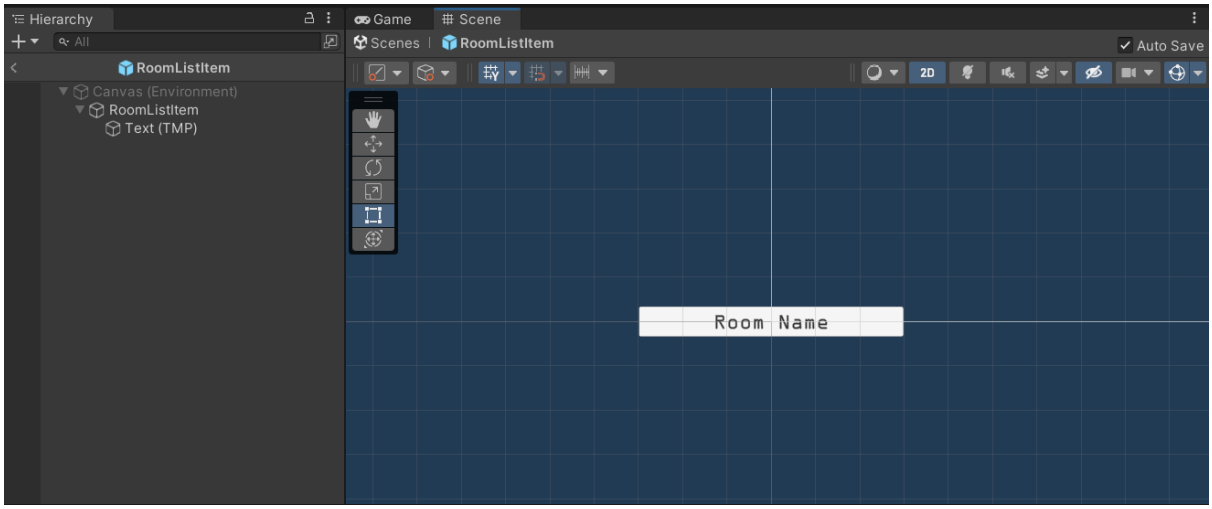






Any prefabs that reside in these menus (such as *PlayerListItem* and *RoomListItem*) are also edited to fit this overall design:

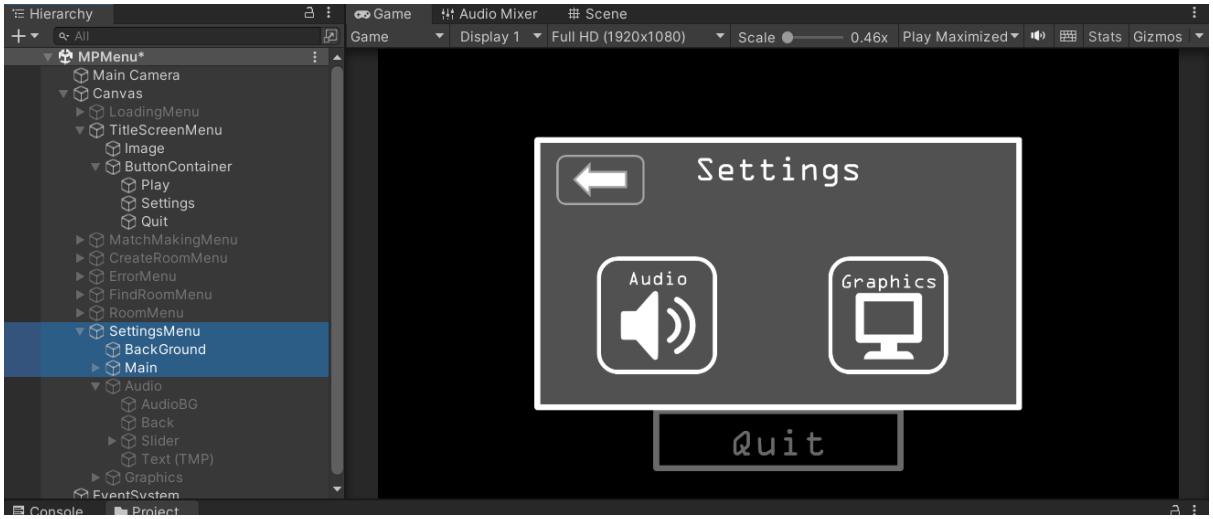




Settings Menu

Now, a settings menu must be added to allow the player to change certain client settings. This will not be a typical menu like all the other ones, but rather an over-lay menu that goes over any menu it is active in.

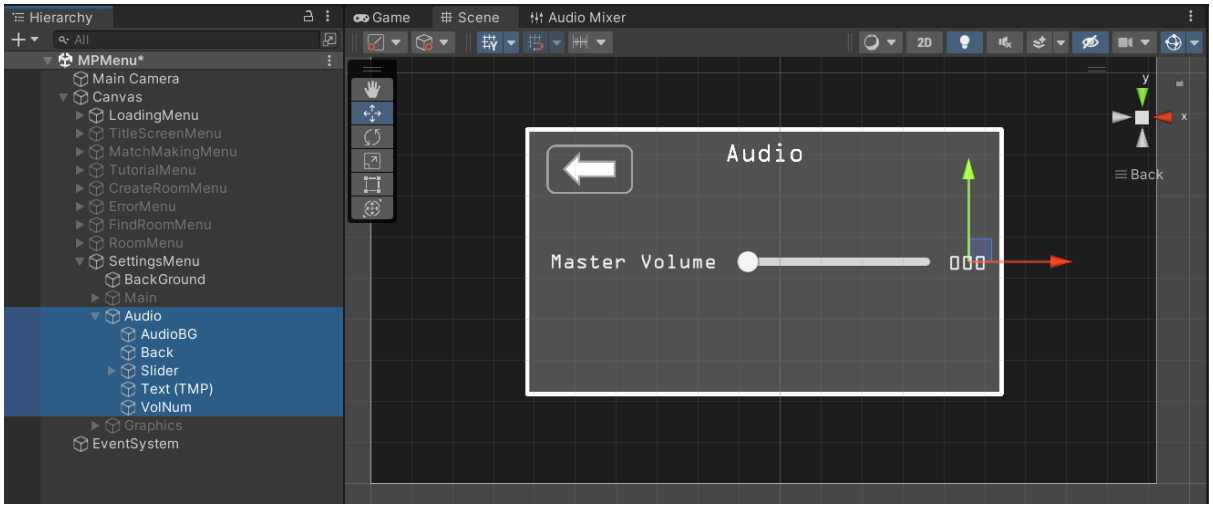
To start off, I make a background image that dims the whole scene slightly for better contrast between the setting menu and the menu it overlays. Then, I add an empty main settings menu GameObject, holding UI elements mimicking the images set in my UI design of the menus:



Then, I make two more menus that will activate upon clicking their corresponding buttons in the main settings menu: Audio and Graphics

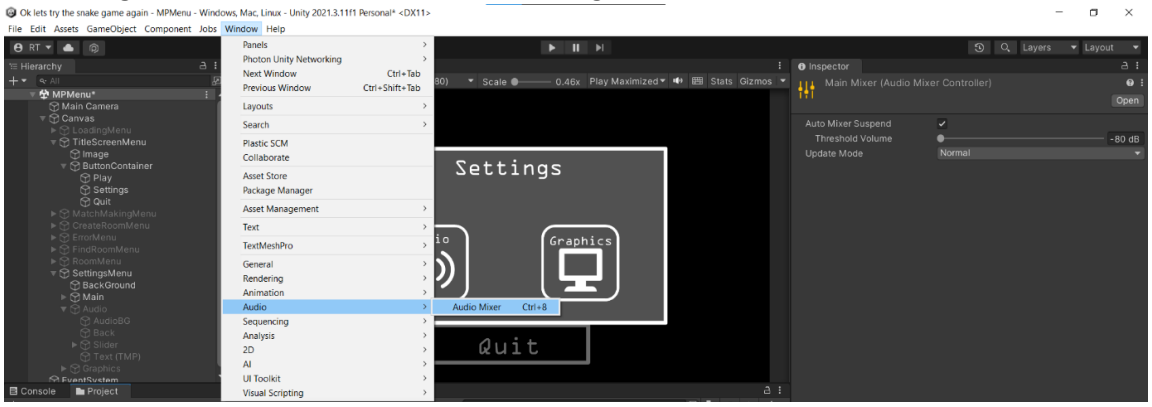
Audio

This menu will handle the volume control of the game. To do this, a background must be first added to hold any settings UI elements. Then, a slider is added with the label “Master Audio” next to it, and a string showing the change of volume as a number is also added:

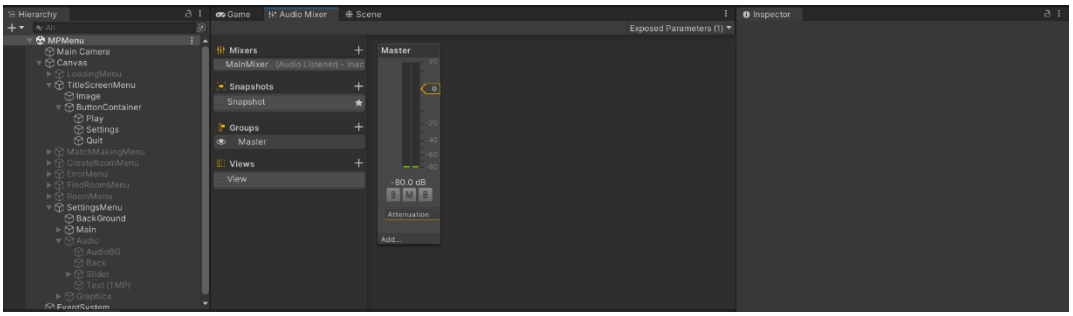


To control how this slider changes volume, a new script is added within the main *SettingsMenu* GameObject also named *SettingsMenu*:

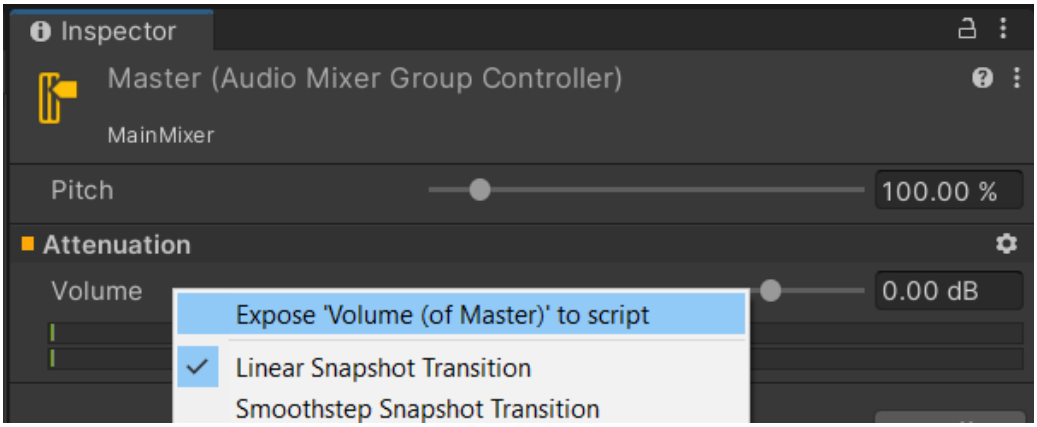
We also need to add a new Audio Mixer to our game, which gives us the ability to control the volume of parts, if not all, of our game. This is done in the Unity Editor by going through the top bar and selecting *Window>Audio>Audio Mixer*, or hitting *Ctrl+8*:



Once this window is open, we can make a new mixer by clicking the plus icon next to the Mixers section. This creates the mixer, bringing us to this menu:



In order to actually change the value on the master mixer (increasing this value increases the volume and vice versa), we need to be able to “expose” the volume parameter, allowing it to be changed via script. To do this, we click on the Master group, go to the right where it says Volume, right click, and select “*Expose ‘Volume’ of Master to Script*”:



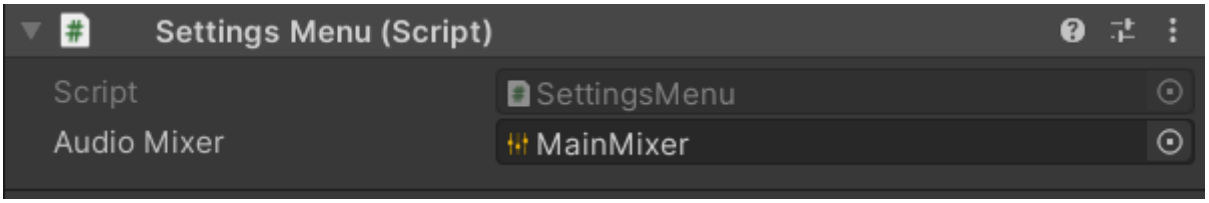
This will indicate to us that the volume parameter is exposed with an arrow next to Volume:

We can rename this exposed parameter by going to the Exposed Parameters tab at the top right of the audio mixer window, right clicking on the exposed parameter and selecting “Rename”. We will rename this parameter to something simple such as “Volume”:

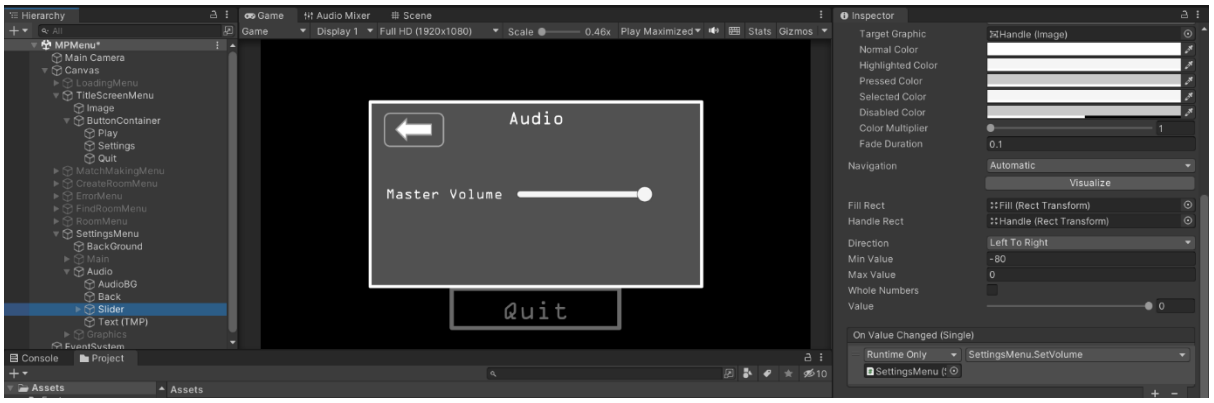
Going back to the *SettingsMenu* script, we can make a new function *SetVolume*, that takes in the float the slider is set at as an argument. We will need to make a new reference to our Audio Mixer (only allowed once the *UnityEngine.Audio* libraries are imported) as a variable *audioMixer*, as well as a reference for the volume text. Then, in *SetVolume*, we set the float of the exposed *Volume* parameter to the float the slider gives us, and the value of the volume text to a percentage out of 100% of the slider value:

```
public TMPro.TMP_Text VolText;


Unity Message | 0 references
void Start()...

// Called whenever the Master Volume slider changes value
0 references
public void SetVolume(float volume)
{
    // Set the volume of the mixer to the slider value
    audioMixer.SetFloat("Volume", volume);
    VolText.text = ((int)(((volume + 80) / 80) * 100)).ToString("000");
}
```

In the Unity Editor we need to assign the *audioMixer* variable to the player's Audio Mixer:



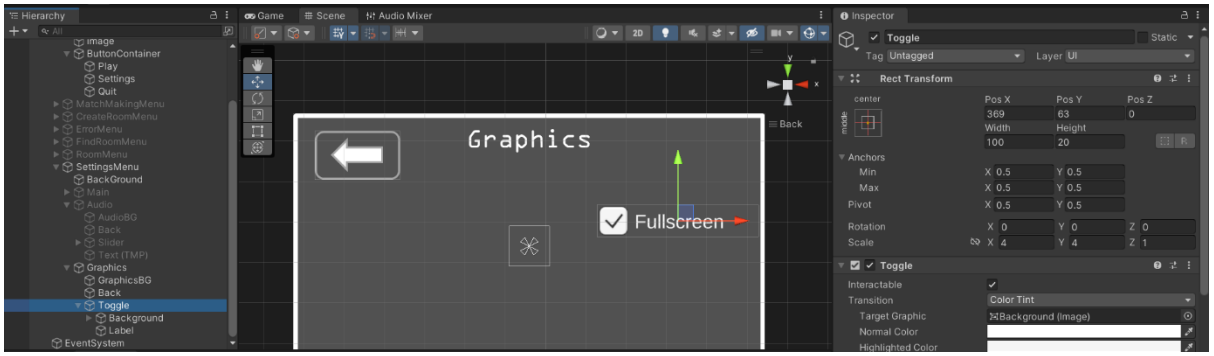
Finally, we need to assign the *SetVolume* function to the slider for it to be able to change the volume:



Graphics

The Graphics menu is set to change the resolution of the game, as well as whether the game is played in Fullscreen or not.

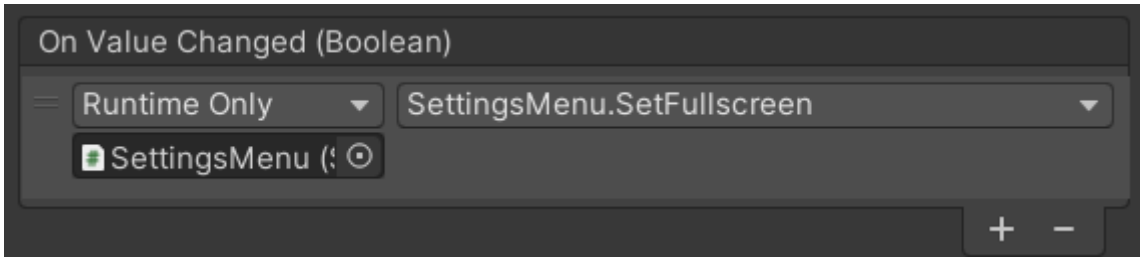
We will start with enabling/disabling Fullscreen mode. In the graphics menu, we will add a UI toggle with the label "Fullscreen" next to it:



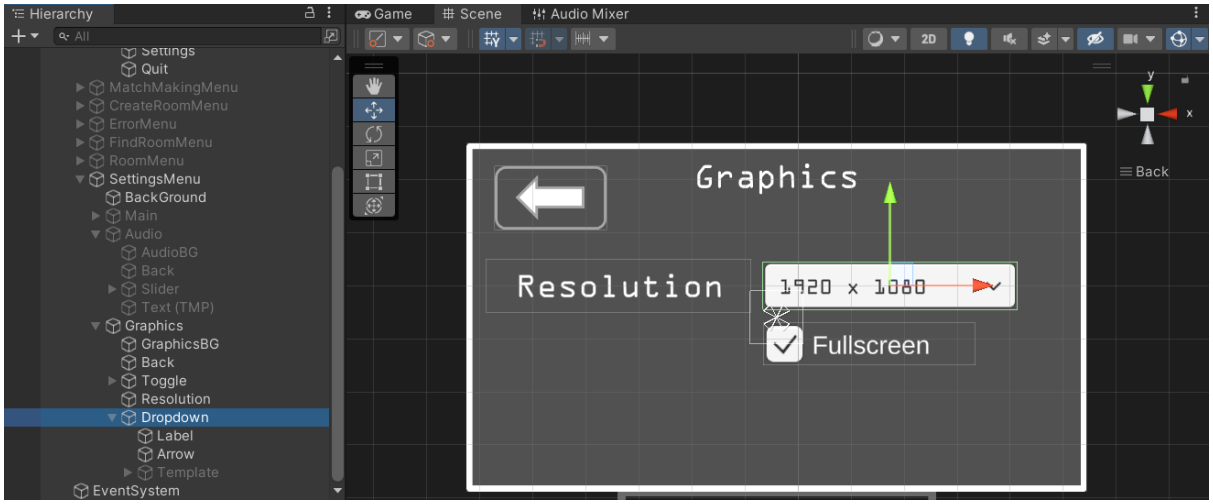
Back in the *SettingsMenu* script, we make a new function named *SetFullScreen*, passing through a Boolean *isFullscreen* as an argument. Here, we set the built-in Boolean enabling/disabling Fullscreen to whatever value the toggle takes when it is clicked:

```
// Called whenever the Fullscreen toggle is clicked
0 references
public void SetFullscreen(bool isFullscreen)
{
    // Set fullscreen to true/false based on the value of the toggle
    Screen.fullScreen = isFullscreen;
}
```

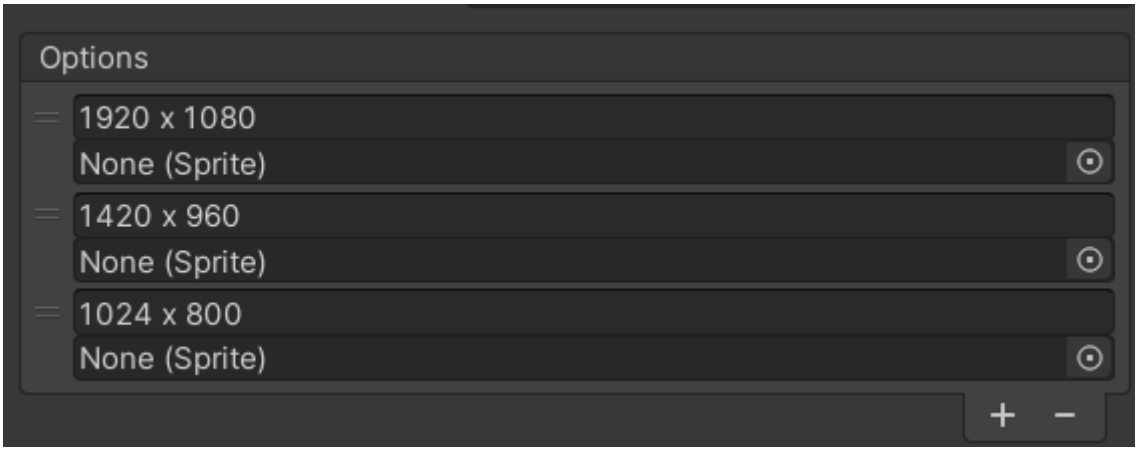
Again, we need to assign this function to the toggle as such:



The final setting we will implement is the resolution changer. The player will be able to change this via a dropdown menu that looks like this:



I also put on placeholder resolution options for the dropdown below:



The reason I say these are placeholders is because the resolutions a player can set their game to differs between machines. Luckily, Unity can figure out what resolutions a certain machine can run a game at.

In the *SettingsMenu* script, we want to gather some information about what resolutions the player has at their disposal as soon as the scene the settings menu is loads, so we put this code in the Start function.

```
Resolution[] resolutions;

Unity Message | 0 references
void Start()
{
    // Grab all resolutions the player can run the game in
    resolutions = Screen.resolutions;
}
```

Here, we assign an array of resolutions to the resolutions Unity finds that are suitable for the player.

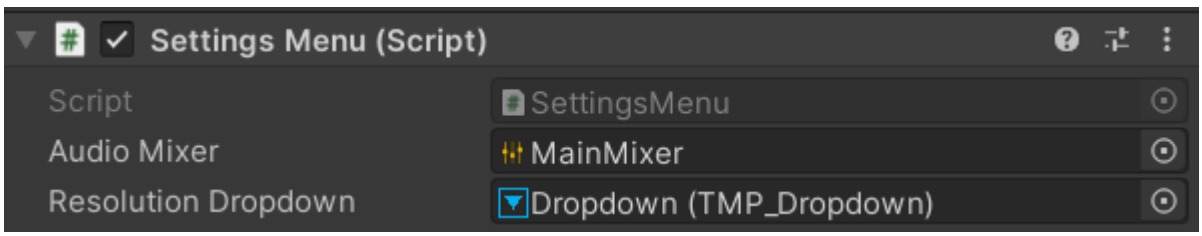
Then, we need to go through and add each of these resolutions to the dropdown menu. To do that, we need a reference to the dropdown menu. After importing *UnityEngine*.UI, we can make a new *Dropdown* variable *resolutionDropdown*. Then, in the Start function, after we grab the player's resolutions, we want to clear the existing dropdown placeholder options, then convert the array of resolutions into a list of strings (as the dropdown menu can only hold string values) before adding this list of strings as values to the dropdown variable:

```
Unity Message | 0 references
void Start()
{
    // Grab all resolutions the player can run the game in
    resolutions = Screen.resolutions;

    // Clear placeholder options
    resolutionDropdown.ClearOptions();

    // Convert the array of resolutions to a list of strings
    List<string> options = new List<string>();
    for (int i = 0; i < resolutions.Length; i++)
    {
        string option = resolutions[i].width + "x" + resolutions[i].height;
        options.Add(option);
    }

    // Add the resolutions list to the dropdown menu
    resolutionDropdown.AddOptions(options);
}
```

This is good for simply getting the resolutions. However, we want the dropdown menu to default to the player's actual resolution. To do this, we find the index of the resolution that is the same as the player's resolution by comparing each resolution's width and height with that of the player. Then, we set the current value of the dropdown menu to that of the player's resolution, then refresh the menu to actually show this value:

```
Unity Message | 0 references
void Start()
{
    // Grab all resolutions the player can run the game in
    resolutions = Screen.resolutions;

    // Clear placeholder options
    resolutionDropdown.ClearOptions();

    int currentResolutionIndex = 0;

    // Convert the array of resolutions to a list of strings
    List<string> options = new List<string>();
    for (int i = 0; i < resolutions.Length; i++)
    {
        string option = resolutions[i].width + "x" + resolutions[i].height;
        options.Add(option);

        // Get the index of the player's actual resolution within the list of resolutions
        if(resolutions[i].width == Screen.currentResolution.width && resolutions[i].height == Screen.currentResolution.height)
        {
            currentResolutionIndex = i;
        }
    }

    // Add the resolutions list to the dropdown menu
    resolutionDropdown.AddOptions(options);

    // Set the dropdown menu's default value to the player's current resolution
    resolutionDropdown.value = currentResolutionIndex;
    resolutionDropdown.RefreshShownValue();
}
```

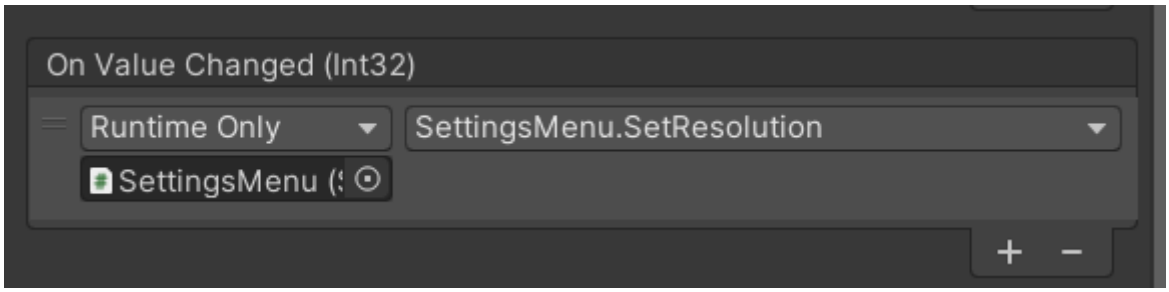
In the Unity Editor we assign the dropdown menu to the *SettingsMenu* script as such:



To update the resolution from this dropdown now, we make a new function in the *SettingsMenu* script named *SetResolution*, taking in an integer *resolutionIndex* as an argument. Then, the code finds the resolution based on the index of it within the resolutions list, then sets the screen to that resolution:

```
// Called whenever the dropdown menu changes value
0 references
public void SetResolution(int resolutionIndex)
{
    // Set the screen to the chosen resolution
    Resolution resolution = resolutions[resolutionIndex];
    Screen.SetResolution(resolution.width, resolution.height, Screen.fullScreen);
}
```

We then set this function to the dropdown menu as such:

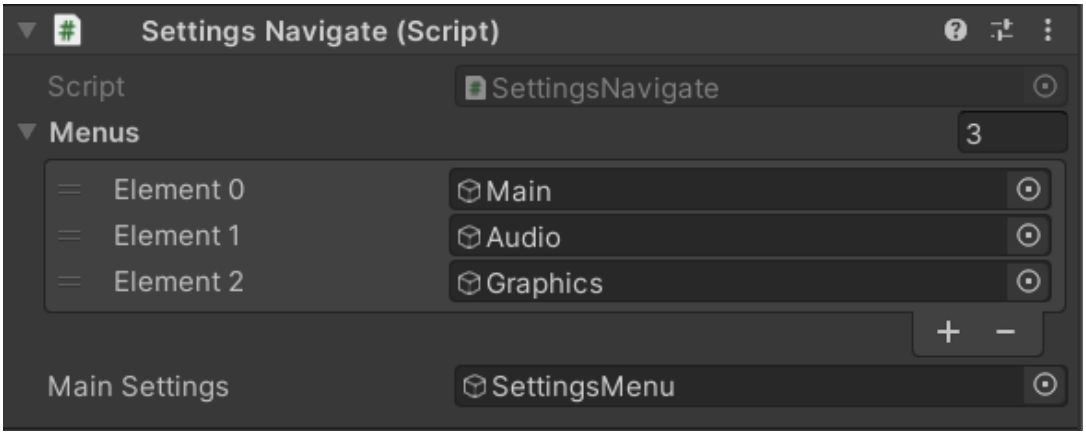


Now that all settings have been accounted for, there should be a way to navigate between these menus. To do this, a new script is added to the UI canvas holding all these menus named *SettingsNavigate*.

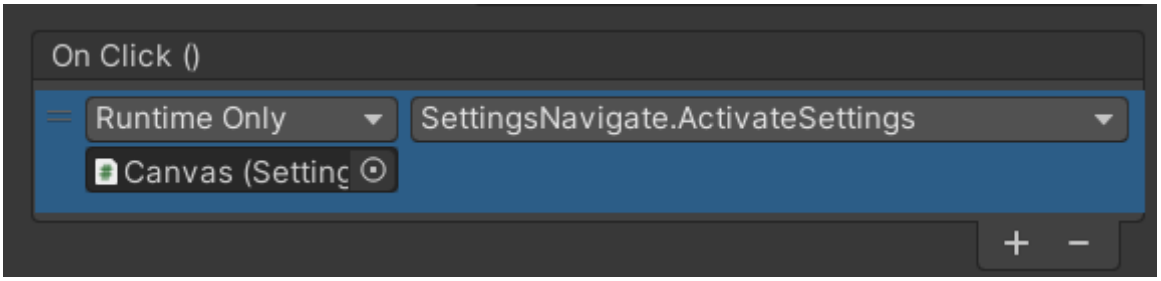
Within this script, references to the *SettingsMenu* GameObject (on its own), as well as all sub-menu GameObjects (as a list) are referenced, and a group of functions are made that either open the main settings menu, close the main settings menu, or open/close the sub-settings menus:

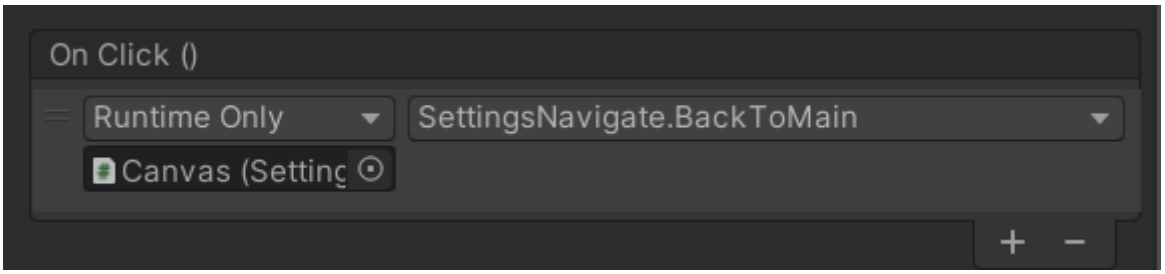
```
public class SettingsNavigate : MonoBehaviour
{

    public List<GameObject> menus;
    public GameObject mainSettings;

    0 references
    public void ActivateSettings()
    {
        // Open the main settings menu
        mainSettings.SetActive(true);
        menus[0].SetActive(true);
    }

    0 references
    public void DeactivateSettings()
    {
        // Close the main settings menu
        mainSettings.SetActive(false);
    }

    0 references
    public void SwitchSettingsMenus(int menuIndex)
    {
        // Close all sub-settings menus and open the menu corresponding to the provided index
        foreach (GameObject menu in menus)
        {
            menu.SetActive(false);
        }

        menus[menuIndex].SetActive(true);
    }

    0 references
    public void BackToMain()
    {
        // Close all sub-settings menus
        foreach (GameObject menu in menus)
        {
            menu.SetActive(false);
        }
    }
}
```

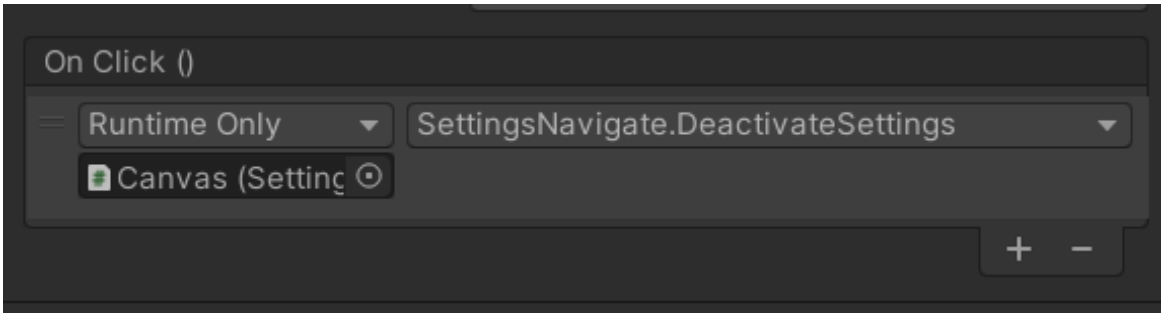
We assign the given settings menus to the script as such:

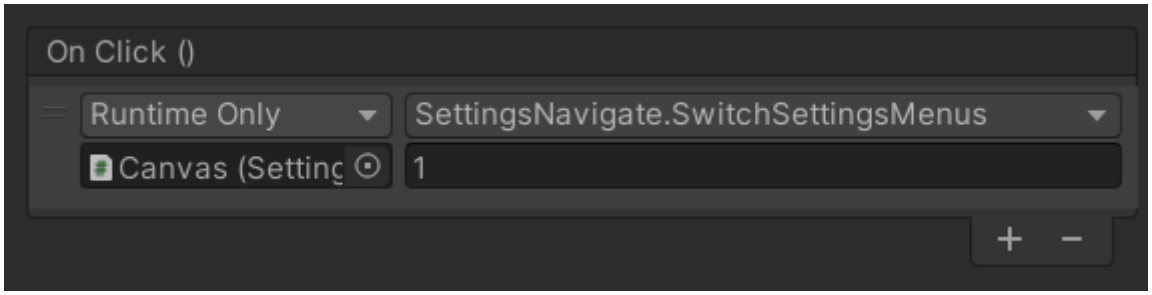


We then assign every navigation button to and from the settings menus to open their respective menus:







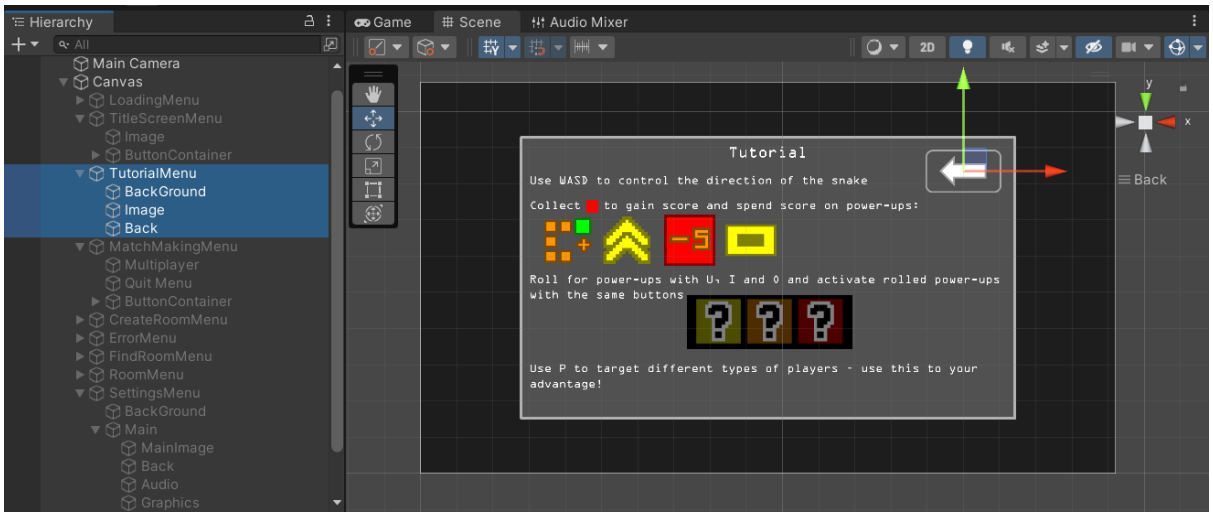


These settings also need to be implemented in the *PlayerController* prefab, in which the same processes are done.

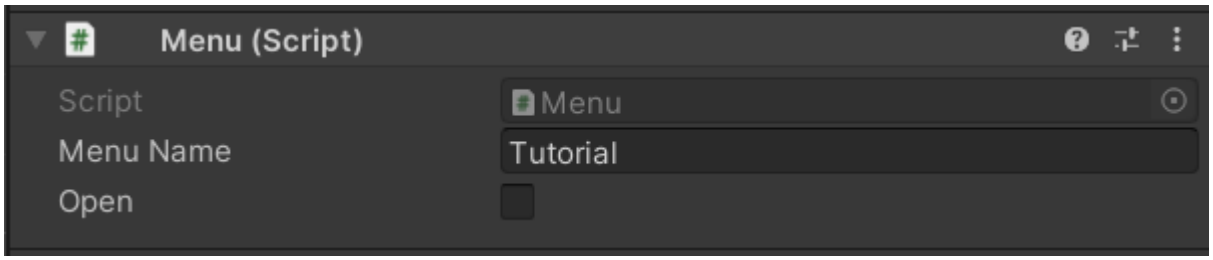
Tutorial

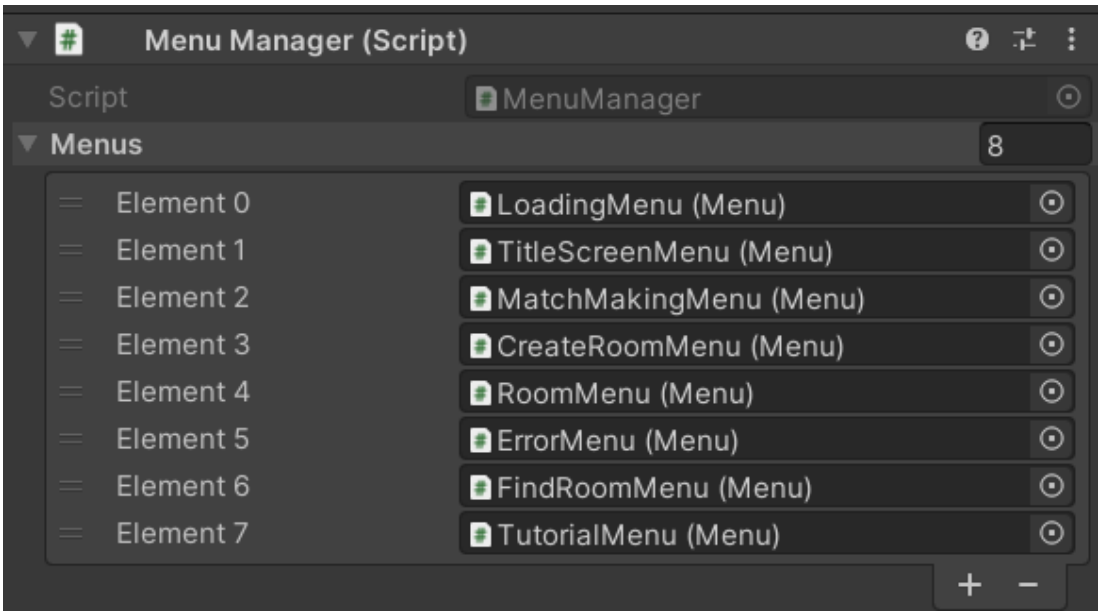
The final menu I will implement is a tutorial menu, which is a simple explanation of how to play the game.

To create this menu, I create an empty GameObject that holds an UI element containing the explanation, as well as a back button to return to the *MatchMaking* Menu:

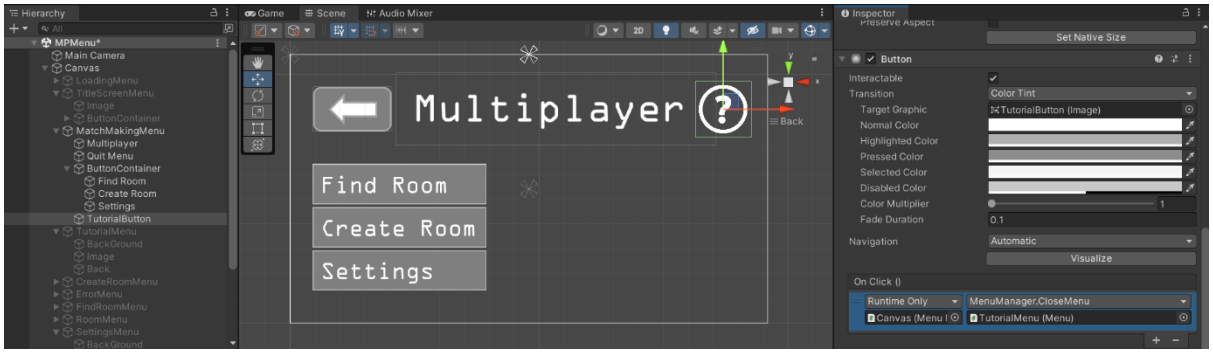


This menu is given its own Menu script, and is added to the list of menus within the canvas:

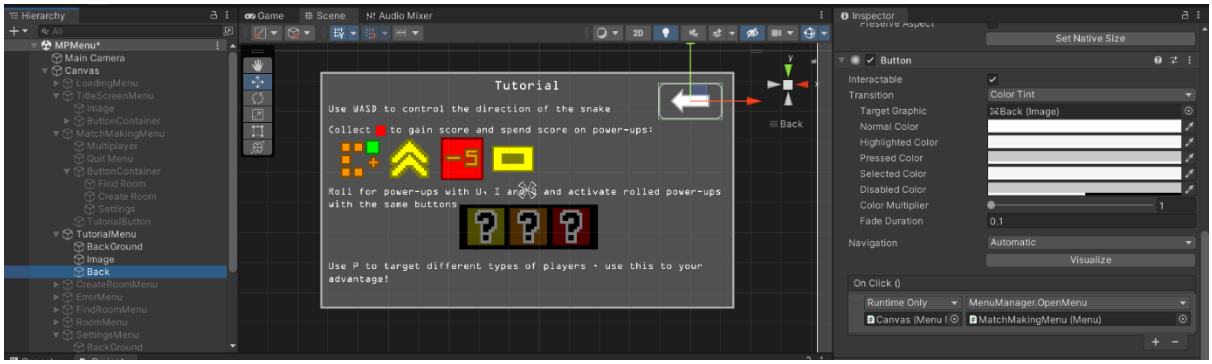




Then, to access this menu, an icon in the MatchMaking menu is made that opens the TutorialMenu via the *MenuManager* script:



Once in the menu, the back button is assigned a function to return to the MatchMaking menu:



Testing Current Project

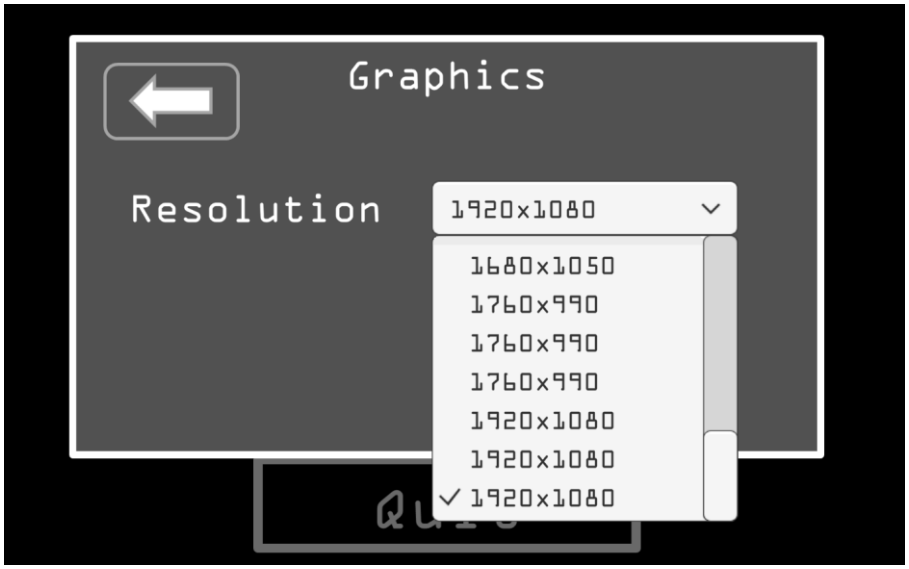
Testing these functions are done within a built version of the game, as they can allow full functionality of the settings menu.

Testing is done as follows:

Test	Expected Outcome	Actual Outcome
Clicking “Settings” in the Title Screen	Settings Menu Opened	*PASS*
Clicking “Settings” in the MatchMaking Screen	Settings Menu Opened	*PASS*
Clicking “Settings” in the Room Menu	Settings Menu Opened	*PASS*
Clicking “Settings” in the Spectating Screen	Settings Menu Opened	*PASS*
Clicking “Audio” in the Settings Menu	Audio Menu Opened	*PASS*
Clicking “Graphics” in the Settings Menu	Graphics Menu Opened	*PASS*
Adjusting the Volume slider	Volume Number Text changes	*PASS*
Clicking the Back arrow in the Audio Menu	Settings Menu Opened	*PASS*
Clicking the resolution dropdown menu	Resolutions the player can use are shown	*PASS*, but there are duplicates of duplicates shown
Clicking an option in the resolution dropdown menu	The resolution of the game changes	*PASS*
Clicking the Fullscreen toggle	The game toggles on/off Fullscreen	*PASS*
Clicking the Back arrow in the Graphics Menu	Settings Menu Opened	*PASS*

Clicking the Back arrow in the Settings Menu	Settings Menu Closed	*PASS*
Clicking the question mark icon in the MatchMaking screen	Tutorial Opened	*PASS*
Clicking the back arrow icon in the Tutorial screen	MatchMaking Menu opened	*PASS*

The only issue found here was that when opening up the list of resolutions on the dropdown menu, there were quite a lot of duplicates:

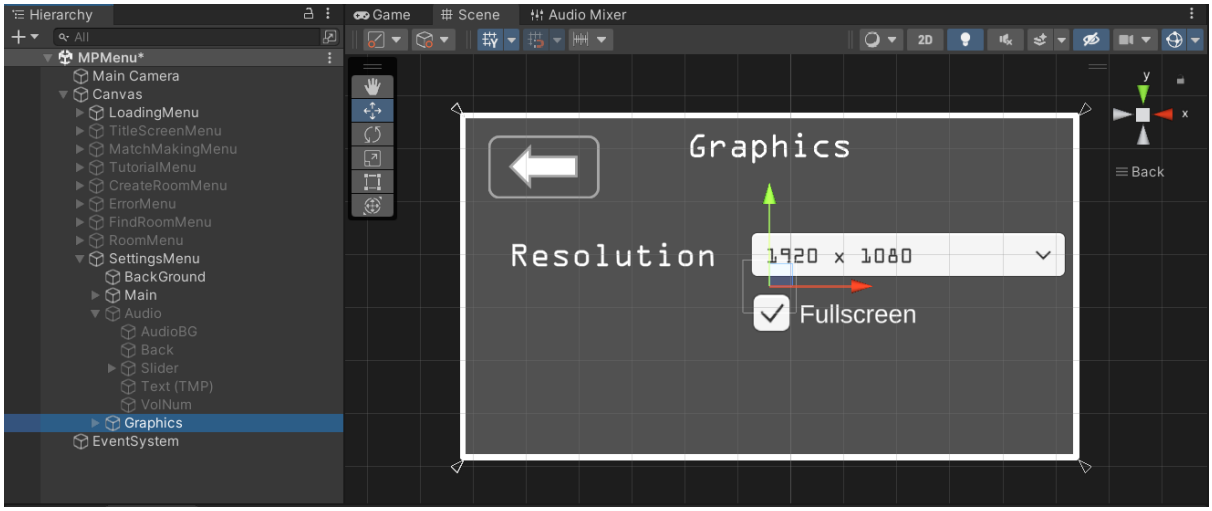


This is because Unity has a tendency to provide options of the same resolution at differing refresh rates. For example, “1920 x 1080 @ 144Hz” and “1920 x 1080 @ 60Hz” both have the same resolution, but different refresh rates. However, as the dropdown menu only holds resolutions, the options look visually the same.

To fix this, we add code to the option string within the *SettingsMenu* script to add the refresh rate to each option:

```
Unity Message | 0 references
void Start()
{
    // Grab all resolutions the player can run the game in
    resolutions = Screen.resolutions;

    // Clear placeholder options
    resolutionDropdown.ClearOptions();

    int currentResolutionIndex = 0;

    // Convert the array of resolutions to a list of strings
    List<string> options = new List<string>();
    for (int i = 0; i < resolutions.Length; i++)
    {
        string option = resolutions[i].width + "x" + resolutions[i].height + " @ " + resolutions[i].refreshRate + "hz";
        options.Add(option);

        // Get the index of the player's actual resolution within the list of resolutions
        if(resolutions[i].width == Screen.width && resolutions[i].height == Screen.height)
        {
            currentResolutionIndex = i;
        }
    }
```

We also need to expand the dropdown menu box to allow for each option to fit without overlapping with the added text:



Now, retrying that test:

Test	Expected Outcome	Actual Outcome
Clicking the resolution dropdown menu	Resolutions the player can use are shown	*PASS*

The testing done here can be seen as a video below:

Final Testing and Evaluation

In this section, I intend to determine how successful my project is. I will do this by comparing the current state of the project with both the success criteria and current opinions of stakeholders. I will determine what I have accomplished well, what I failed to accomplish, as well as what I can improve upon in terms of features my project has.

Usability Testing and Evaluation

Here, I will get stakeholder feedback for my project. The reason I wasn't able to do this throughout my development process is because each iteration built towards a functioning copy of a game, and only at the point where the game is properly functioning is where I can show this to my stakeholders.

After testing the game with my stakeholders, I asked them how they felt about the overall functionality of the game. Their responses were as follows:

- "The UI of the game looks really nice, and gameplay feels relatively smooth (although for someone with slower Wi-Fi that's probably not going to be the case). The tutorial explains

the game well (although it does take some time getting used to who you are and who everyone else is). I really like the ability to change the resolution of the game, but I will say, disabling Fullscreen at something that is not 1920 x 1080 does end up with a few UI buttons and whatnot being in random places. Overall, I think this is a useable project and I had fun playing it!"

- "I think the buttons and style of the game is very nice, but there should be a way to change the background of the game, for example, just because the game can look bland at times. The game itself is really fun, but once the game is started, there is no countdown or anything to delay the players before they start moving, which may cause players to die easier. Also, some sound effects to the game would be nice, because playing this game without any sound at all feels very eerie. Overall however, nice game."
- "The game has little breathing room between starting the game and moving, which absolutely is needed because I sometimes die far too quickly without knowing. Movement is relatively responsive, but I feel it could do better with some sort of buffer system that stores inputs if multiple are put in at the same time. I liked the general aesthetic of the game and how it can be perceived as minimalistic. I also like how the User Interface is structured, giving you many opportunities to change settings if needed. Overall, I like this game apart from the few issues I had with it."
- "The simplistic menus are clearly labelled which helps with the user experience. Settings do exactly what I need, with audio and display options allow me to have the game experience cater to my needs. The use of graphics also makes it very visually appealing. Gameplay is responsive and clean - the game allows me to control my snake character with as much dexterity as I need, which is particularly important in a fast-paced game such as Snake. The graphics looks very nice, and I really enjoy the theme going on. The multiplayer aspect is a twist on Snake that I have never seen before, and I love that I can put my Snake skills to the test in a competitive setting which fulfils my competitive thirst. One improvement I would like to see in the future is a countdown from the start of the game. The game throws you in as soon as it starts, giving the player limited time to prepare. Overall the game is excellent and a very unique version of Snake that I thoroughly enjoy playing, with the best aspect being the ability to battle other players."
- "The tutorial gives clear instructions for the user and tells me exactly how the game works. One small point would be that the audio defaults to 0, meaning that users who are starting up the game for the first time will unknowingly be playing on mute. The powerups add complexity to the gameplay and overall make it more fun to play. However, it is a bit hard to tell which screen is the one you should be looking at. A marker of some sort would be perfect for this. In addition, power-ups should come with some sort of warning to whoever is receiving it, to give them time to prepare. For example, the small walls power-up is overpowered because the player can just run into it without warning. Apart from these issues, I found the game very enjoyable!"

From these responses, I can tell that, generally, movement and UI were the main strengths of my project. However, a particular weakness was a lack of warning before some actions occurring (such as spawning and sending power-ups), which, upon viewing the test videos, was an issue I completely overlooked. Overall, however, the game seemed to satisfy their requirements and was generally accepted as a useable project.

Function and Robustness Testing

Here, I will compare parts of my project with the success criteria I have set for myself.

Criteria	How to evidence	Completed?
An easy-to-navigate Interface	https://youtu.be/z7sAmaa_u18	FULLY
Evaluation: The evidence shown above shows that the user is clearly able to navigate through these menus with either text buttons clearly stating the menu they go to, or icons indicating the menu the player will enter/exit. Icons clearly express their purpose and all buttons give a aesthetically pleasing colour change when hovered over/clicked.		
A way to allow the user to configure in-game sound	0:43 at https://youtu.be/z7sAmaa_u18	FULLY
Evaluation: The evidence shown above shows that the user is clearly able to adjust the volume of their game using the slider provided to them. In addition, an integer value provided next to the slider can allow for more precise control over the volume, as the user can set the volume to be a specific value.		
A way to allow the user to configure graphics such as resolution	0:53 at https://youtu.be/z7sAmaa_u18	FULLY
Evaluation: The evidence shown above shows that the user is clearly able to alter the resolution of the screen to their liking from the dropdown provided, as well as check if the game will be Fullscreen or not. The user is also able to keep these choices when exiting the settings menu.		
A way to allow the user to configure controls	N/A	NOT MET
Evaluation: There is no evidence to provide for this criteria as it has not been met. This is due to a lack of time to implement this within my project, and was a consequence of me not being able to schedule what I am able to do within my project time. If I was to make this project again, I would make this menu to allow the user to customise controls in order to make them more comfortable to the user. This can be done with Unity's own input system.		
A menu that allows the user to customise the looks of the snake, food and the background	N/A	NOT MET
Evaluation: There is no evidence to provide for this criteria as it has not been met. This is due to a lack of time to implement this within my project, and was a consequence of me not being able to schedule what I am able to do within my project time. If I was to make this project again, I would make this menu to allow the user to customise themselves more, and add more reason to play the game (to gain cosmetics)		
Progress Tracking with a Progression System	N/A	NOT MET
Evaluation: There is no evidence to provide for this criteria as it has not been met. This is due to a lack of time to implement this within my project, and was a consequence of me not being able to schedule what I am able to do within my project time. If I was to make this project again, I would make this menu to allow the user to visualise their progress in-game, and add more reason to play the game (to gain more levels)		
A way to enter a matchmaking lobby	0:20 – 0:37 at https://youtu.be/z7sAmaa_u18 and	FULLY

Evaluation: The evidence shown above shows that the user is able, through good menu navigation, to take themselves to create a room. Evidence also shows the ability to find a room that has been created (shown by an updating list of available rooms), and the ability to join that room, as long as it is not full or the game has not started.		
A way to spawn players in separately	0:06 at https://youtu.be/eG7GhUggUYA	FULLY
Evaluation: The evidence shown above shows that the user is able to, upon the game starting, have their snake, food and walls spawn in their relative place. The player is also able to see all other players have their snake, food and walls spawn in their relative spawn points away from the local player. UI items for the player only seem to appear on that player's side and not anyone else, which is a good thing.		
Real-time multiplayer networking	https://youtu.be/eG7GhUggUYA	FULLY
Evaluation: The evidence shown above shows that the user is able to view the movements of other players as they happen (as long as all devices have a solid internet connection), and actions performed between devices are synced to all players properly. For example, if a player changes direction, all players will see that player change direction, and when they activate a power-up, all players will see that in action.		
A way to change the direction of the snake using user input	https://youtu.be/Sa_fntGKLcY and https://youtu.be/eG7GhUggUYA	PARTIALLY
Evaluation: The evidence shown above shows that the user is able to change their snake's direction by inputting the directional keys, and see a visual change of this within the game. However, there are times where, inputting two directions in quick succession results in the snake moving into itself, resulting in the death of the snake. This is due to the coroutine controlling the direction, in which the indicator of direction can be changed twice before the snake moves one unit. If I was to produce this project again, I would make a system that stops the player making any directional input if the final direction inputted before the snake moves is the opposite of its current direction.		
Detection for when a snake collects a food item and for the score of the user to increase accordingly	https://youtu.be/Sa_fntGKLcY	FULLY
Evaluation: The evidence shown above shows that the user's snake is able to collide into food, and for that snake to grow one segment, as well as the player to gain one point of score as a result.		
Sound/Visual effects when the Snake obtains food	https://youtu.be/Sa_fntGKLcY	PARTIALLY
Evaluation: The evidence shown above shows that, when the user's snake collides into food, a small particle effect is instantiated and played. However, no sound effect is played, due to a lack of time to be able to implement this. If I was going to make this project again, I will be able to add sound effects for this occurrence.		
Detection for when the snake collects a certain number of food and for the	https://youtu.be/Sa_fntGKLcY	PARTIALLY

speed of the snake to increase accordingly		
Evaluation: The evidence shown above shows that, when the user's snake collides into 5 pieces of food, there is a visible increase in speed of the snake. However, there is no indicator that the speed has changed for the player other than in the snake itself, which is something I will do should I recreate this project.		
A way to target different players	2:28 at https://youtu.be/Sa_fntGKLcY	FULLY
Evaluation: The evidence shown above shows that, when the user presses a certain key (P), the UI for indicating what form of targeting there is changes, and the targeting function itself is working, as there is a red outline around the walls of the target player.		
A way to obtain powerups using user input	1:10, 1:46 and 2:06 at https://youtu.be/Sa_fntGKLcY	FULLY
Evaluation: The evidence shown above shows that, when the user gains 10, 15 or 20 score, they are able to spend them on random rolled powerups with either U, I or O, and end up having an icon for the powerup to use. If I was to make this project again, I would add more powerups to provide more variety in gameplay for the user.		
Sound/Visual effects when the Snake obtains a powerup	1:10, 1:46 and 2:06 at https://youtu.be/Sa_fntGKLcY	PARTIALLY
Evaluation: The evidence shown above shows that, when the user rolls for a power-up, a powerup randomising effect is played (the icons are randomised for one second). However, no sound effect is played, due to a lack of time to be able to implement this. If I was going to make this project again, I will be able to add sound effects for this occurrence.		
A way to send the effects of a powerup to other "alive" players in game	2:09 at https://youtu.be/Sa_fntGKLcY, and 1:26, 2:20, 2:56 at https://youtu.be/eG7GhUggUYA	FULLY
Evaluation: The evidence shown above shows that, when the user gains a powerup and press U, I or O based on which slot the powerup is in, they are able to use them on other people or themselves. However, if I was to recreate this project, I would also add to it the ability for a countdown delay before a powerup activation to give players more time to prepare.		
Sound/Visual effects when the Snake uses a powerup	2:09 at https://youtu.be/Sa_fntGKLcY, and 1:26, 2:20, 2:56 at https://youtu.be/eG7GhUggUYA	PARTIALLY
Evaluation: The evidence shown above shows that, when the user uses a power-up, the powerup is used up, indicated by its icon disappearing from the user. However, no sound is played here. This was unable to be coded in due to a lack of time. If I was going to make this project again, I will be able to add sound effects for this occurrence.		
Detection for when the snake hits an obstacle and for appropriate loss visual/sound effects to appear	2:34 at https://youtu.be/Sa_fntGKLcY, and https://youtu.be/eG7GhUggUYA	FULLY
Evaluation:		

The evidence shown above shows that the user's snake is able to collide into either the walls (given that it is not NoWalls) or itself, and for that snake to stop moving and delete all segments, as well as displaying a Game-Over screen for the player. However, upon checking the game as a whole, I realise that perhaps more effects could be played to make the death of the player more visually appealing and not just instant. Perhaps adding a time delay between the player's death and any extra UI popping up may add space to create a death animation for the snake		
A way to notify players when one player dies	0:15, 1:50, 4:54 5:27 and 7:28 at https://youtu.be/eG7GhUggUYA	FULLY
Evaluation: The evidence shown above shows that all players are notified of the death of a player via the K.O indicator they display when they die, and that the player actually displays this indicator to everyone when they die.		
A way to notify players when they get a kill	0:37 at https://youtu.be/hZwxV8Ru9C0	FULLY
Evaluation: The evidence shown above shows that the player that killed the player who died gets an indicator of a kill via the kill count they have increasing by 1. This is also displayed when they win/lose.		
A way to notify players when one player wins	0:43 at https://youtu.be/hZwxV8Ru9C0	FULLY
Evaluation: The evidence shown above shows that all players are notified of the win of a player via the win indicator they display when they win, and that the player actually displays this indicator to everyone when they win.		
A way to notify the user of their in-match position placement, as well as how many eliminations they get	2:34 at https://youtu.be/Sa_fntGKLcY, and	FULLY
Evaluation: The evidence shown above shows that the user, upon winning or losing, will get the number of kills they have made as well as the placement within the room. Both of these sections are in separate areas and are coloured differently to make them easier to read and differentiate from each other.		
A way to provide new users with a tutorial on how to play the game	0:10 at https://youtu.be/z7sAmaa_u18	FULLY
Evaluation: The evidence shown above shows that the user is able to see how to control their player, as well as the overall aim of the game, through the tutorial menu. Visual aids help with these explanations.		

Maintenance

My project could be expanded upon by adding more power-ups. This can be done with relative ease:

- Adding a new class for the power-up if is a type that is different from existing power-up types.
- Adding the new objects for the power-up and filling them in within the Editor
- Making new sprites for the power-up with GIMP
- Adding new checks in the *PlayerItems* script, as well as functions/RPCs for them in the *PlayerController* script.
- Assigning the new power-up in any scripts that have lists of power-ups

I would also add more players per room. This can be done by:

- Adding more spawn points for *PlayerControllers* to be instantiated
- Increasing the limit of the number of players in a room

I would add new menus to customise the player more, as well as view progression for them. Allowing for the navigation between this menu and others are simple:

- Make the menu (and assign it a Menu Script, as well as adding it to the list of menus within *MenuManager*) as well as any buttons needed.
- Assign buttons that take the player to existing menus to open those menus using the *OpenMenu* functions on the *MenuManager*)

However, implementation of functions within each menu will be more difficult, as this requires access to and knowledge of different functions.

For example, with customisation menus, I would need to find a way to display the player's choice of skin to every player and sync it. This will have to be done using Custom Properties, a function set that Photon provides, that allows for this form of syncing.

Finally, I would add an extra mode allowing the player to practise the game on their own, without multiplayer. This is simple enough, as all I will need to do is implement the base gameplay without multiplayer networking, and will only need to use functions that the local player can run, rather than RPCs.

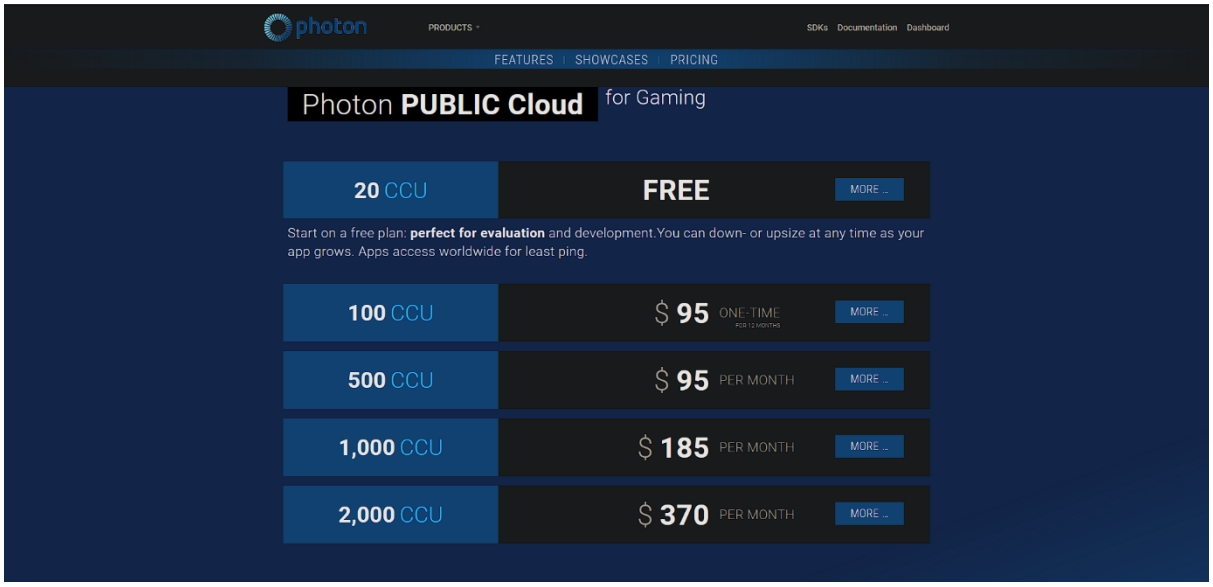
Limitations

Visualising the player in the centre of the screen

One idea for the design I had for this game was being able to expand and display the user's play area at the centre of the screen. Typically, this would be done by making a UI element containing a RawImage component, and having a camera that displays its output to the RawImage via a render texture. However, the canvas that I hold all my UI elements in uses a render mode of "Screen Space – Overlay". This means that Render Textures do not get rendered here, and will not do so until the render mode of the canvas is moved to "Screen Space – Camera". As my *PlayerController* prefab has no camera to capture the whole game space (the camera is in the main game scene, which the prefab cannot access via the inspector), this option is not available, and thus this process of capturing the player's play space and expanding it is not possible. However, as there is still an indicator of who the local player is in a game, this limitation is not a large issue.

The number of players in a server

Expanding the game means allowing more players in a server. However, this proves to be quite a hinderance, due to the networking solution I am using, Photon Unity Networking (or PUN for short) 2 has two tiers: PUN 2 Free, which only supports up to 20 concurrent players in a server, and PUN 2 Plus, which has added to it a 12-month subscription to the 100 concurrent user plan for Photon Cloud. As suggested by the names, PUN 2 Plus is a paid service (for $95), and will only give the ability to host for 100 players maximum for 12 months. After this point, expanding the game for more players becomes a problem financially: to keep allowing up to 100 players, a yearly subscription of ($95) needs to be made, and plans go up to being able to hold 2000 concurrent players at any one time for a monthly $370 payment or a yearly $3700 payment (albeit, this is only needed if the game gets a monthly active userbase of 800,000 players).



This means that I will not be able to expand my game to many more players without paying what may be a hefty fee.

Hacking

With every online multiplayer game there will be some bad actors who look to hack in the game for various reasons. With a game such as this, it may not be very hard for players to install applications that modify the game in a way that they gain an unfair advantage.

Improvements

Tutorial

While the tutorial does provide a clear walkthrough of how to play the game, it does not seem to be interactive and interesting enough to keep a first-time user entertained. A tutorial that consists of the player, in a single-player environment, experiencing the game first-hand rather than some text would be a lot more interesting and would help the player learn more about the mechanics of the game.

Game UI

The menus themselves do not need too much changing, but there aren't any transitions between these menus. Having transitions can spruce up a game, making it more visually appealing for the user. This can be done in two ways – either using the Unity provided animator to automate and call animations of different UI elements within the Unity editor, or In Betweening (Tweening), which is script-coded animations that get stored as components within UI elements. While having an animator may seem easier at first due to a more user-friendly, less text-based interface, an animator is also incredibly expensive for the game itself, as it runs on every frame. On the other hand, Tweening, while having a bit more coding, is incredibly flexible and can still pack many, if not all, the same features as an animator does, with a lot less processor performance drops. Thus, tweening would be the way to go for Menu UI transitions to look seamless, while not sacrificing computer performance.

Anti-Cheat

To help solve the issue that arises with hacking, anti-cheat measures can be implemented on my project. These measures aim to check if any suspicious programs are running/already in the project files, and if any are spotted, they will act on the user holding these programs. While I could implement my own anti-cheat system, there are a lot of systems already available to me that I can

implement to my project, such as Easy Anti-Cheat, a free software that is used by many prominent games worldwide.

Final Code

https://github.com/Skullshock986/Snake-Remade-Again