Software Yr 12 Major Project Report

Dominick Widjaja

Table of Contents

1. Defining and Understanding the Problem

The problem is to create a commercial or educational product for the purpose of a Year 12 Software Design and Development major project. In order to fulfil the requirements, the solution to this problem will be an interactive computer game which is to be commercially sold on the market. This solution has been chosen as it would allow for a wide range of technical expertise to be displayed and contain enough material to be included in the following project report. The interactive game is planned to be a golf simulator from a 2D top-down perspective with replayability and progression in mind. The game will be made using the tools available in the popular game development engine and environment, Unity. The development would also be supplemented with the use of Unity Scripts based on the popular coding language, C#, written using the widely known programming environment, Visual Studio.

Requirements Report

This project was chosen as it would allow for a wide range of technical expertise to be displayed and contain enough material to fulfil the requirements of the assignment. However, this project was also chosen to reflect personal interests and skills related to the project. The best way to accomplish this was through the development of an interactive game. Needs for the project include:

- Functional and playable for players of any skill
    - The inputs were kept simple so that all players are able to interact and enjoy the game. Additionally cultural, social, gender, and disability inclusivity were kept in mind to allow all users equal access to the software.
- Multiple courses/stages with varying themes and unlockable items
    - These stages showed the natural progression of the player's journey and also the underlying story which would supplement the player's enjoyment of the game.
- Customisability in regards to the application and features within the game
    - Players are able to customise their golf balls as well as the resolution, screen type, and audio levels to suit their liking.

While not an extensive list, the dot points outline the features that are required for the project to be considered a success. However, as the scope of the project allowed, the following features were also added:

- Some method to encourage replayability (high scores, time trials, etc)
    - Best Times and Best Hits are stored by the game and the player is encouraged to beat their old records. Players are also able to challenge their past attempts to analyse their past plays against a Ghost Player.
- Local multiplayer
    - A way to greatly increase the enjoyability of the game as users are able to invite their friends and competitively challenge each other for best times.
- Support for various input types (keyboard, controller, etc)
    - In addition to customisability, this would allow for greater inclusivity as players would be able to choose between using their most comfortable control type.

Fulfilling these requirements would allow for the project to be considered a success and the problem to be solved.

2. Identify a Suitable Development Approach

The approach that will be used for the solution is a combination of the Rapid Application Development (RAD) and Prototyping approach.

The RAD approach is defined as creating a usable solution at the lowest cost while taking the least amount of time. This approach would allow for the reusability of modules and the use of CASE tools which aims to save time and resources during development. Resources such as modules of code, textures, and assistance could be taken from Unity's built-in Unity Asset Store and other sources on the internet while still respecting the Intellectual Property of individual creators. As there is no budget allocated towards this project, it would be ideal to keep costs as low as possible. CASE tools could also be utilised from Visual Studio in order to easily create charts and diagrams such as Data Dictionaries, Context Diagrams, and Structured Diagrams.

The Prototyping approach utilises multiple prototypes as a tool to redefine the solution. As the solution requires a User Interface (UI), this approach would allow for a UI to be developed overtime as the requirements of the solution are better understood with each iteration. As prototypes are quickly and easily developed, it would not remove any time from the actual development of the interactive game. Additionally, this approach allows for the visual component of the UI to be considered in isolation from the underlying code. Therefore, it will allow for a stunning User Experience once the final iteration is achieved.

The Structured approach would not be feasible for this solution as their hefty budgets and lengthy development times would exceed the limited time and limited budget for this major project. Additionally, its use of fully custom solutions and strict adherence to the Software Development Cycle would not work well due to the casual method of development and lack of resources. As this solution is not planned to receive continuous updates after release, the AGILE approach would not be well suited either. In addition, the solution would only be worked on by a single developer so the Structured and AGILE approaches which require multiple teams would be extremely inefficient. While the End User approach remains the most approachable, it's dedicated use of Custom-Off-The-Shelf packages would be quite restrictive and not allow for the level of customisibility that is required with the project, hence requiring some level of technicality that can only be achieved with the RAD approach.

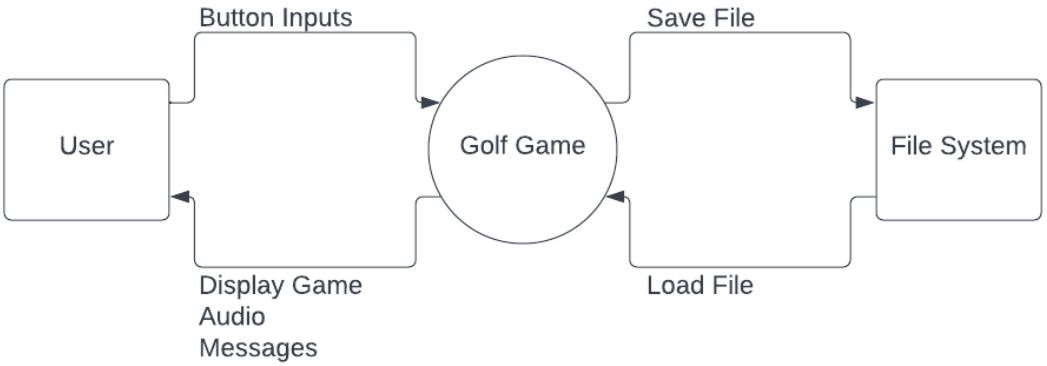
As a result, despite the lack of formality and structure of the RAD approach and the additional skills required by the Prototyping approach, they remain the chosen approaches for the solution as it allows for the requirements of the solution to be completed as quickly as possible, and at least cost, which is required for this solution.

3. Identifying Input, Process, and Output

The following charts and diagrams are vital modules of code that will be required for the final solution.

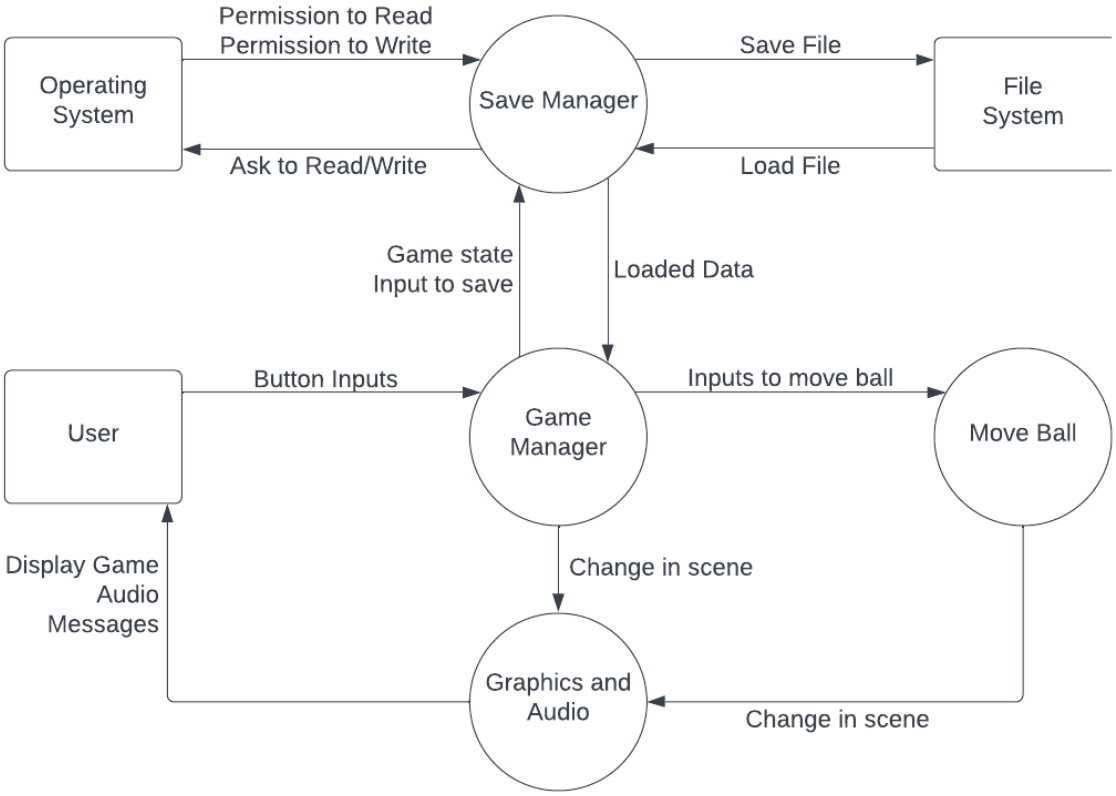
Context Diagram

Detailing the game as a system and the movement of data between processes and external entities.



Data Flow Diagram (Level 1)

More detailed diagram of the processes occurring within the system.



IPO Charts

Charts showing the processing occurring to inputs in order to yield outputs.

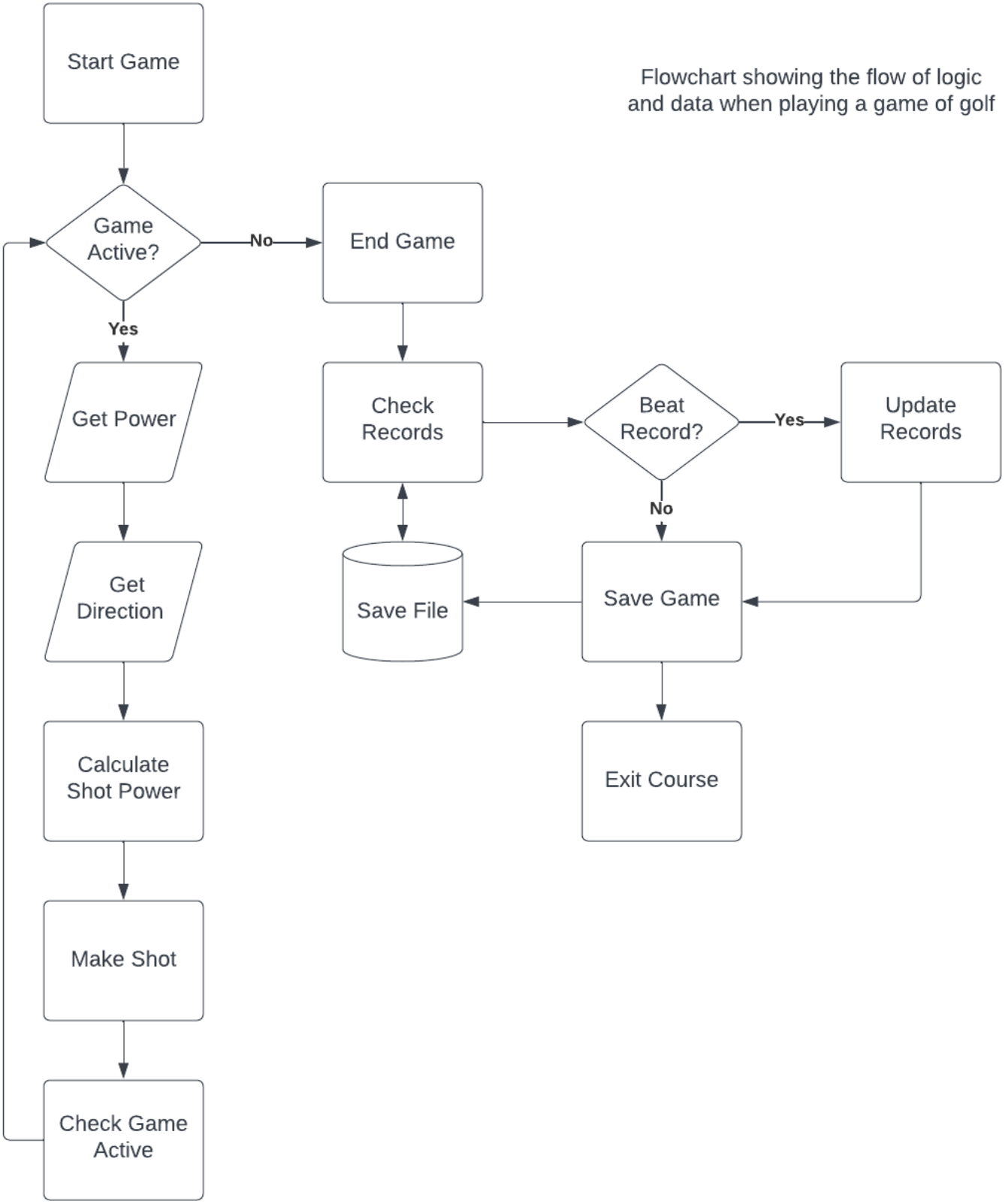
IPO Chart Project: Golf Game Subroutine: AimingModule		Date: 20/11/21
Input	Process	Output
Keyboard Buttons	Check buttons being pressed	
	Rotate direction of aim and change the strength of the shot	
	Send direction and strength of shot to move ball	StrengthOfShot, DirectionOfShot
	Disable inputs until ball stops moving	DisableInput
	Enable inputs when ball stops	EnableInput

IPO Chart Project: Golf Game Subroutine: SaveModule		Date: 20/11/21
Input	Process	Output
SaveCommand	Find all data required to be saved	
GameState	Combine all data into an array of data items	
	Convert data items into binary	
	Open File Access	

	Output binary data to file	SaveFile
	Close File Access	

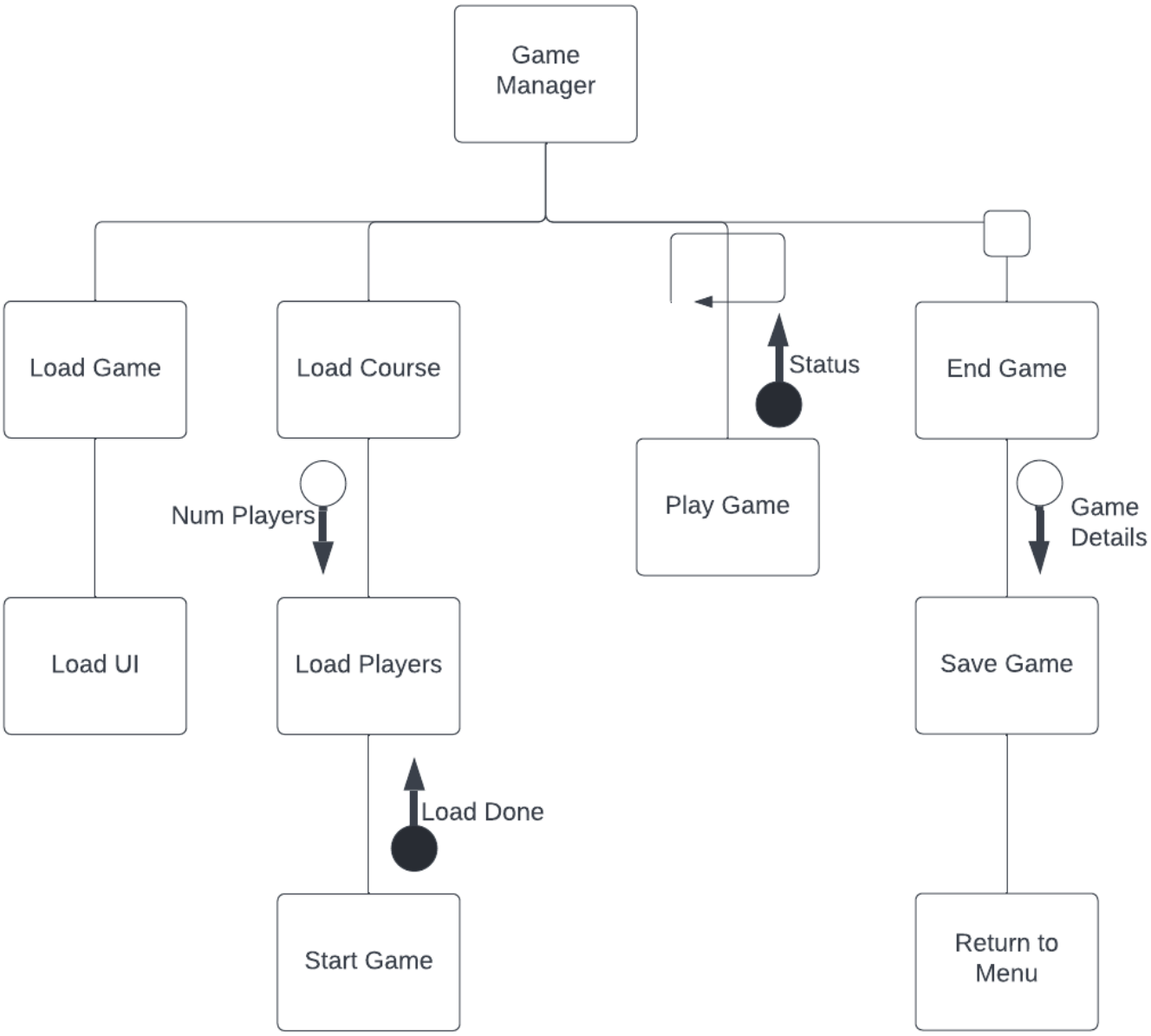
System Flowchart

Showing the flow of data and logic through a system.


Start Game
Flowchart showing the flow of logic and data when playing a game of golf
Game Active?
No
End Game
Yes
Get Power
Check Records
Beat Record?
Yes
Update Records
No
Get Direction
Save File
Save Game
Calculate Shot Power
Exit Course
Make Shot
Check Game Active

Structure Chart

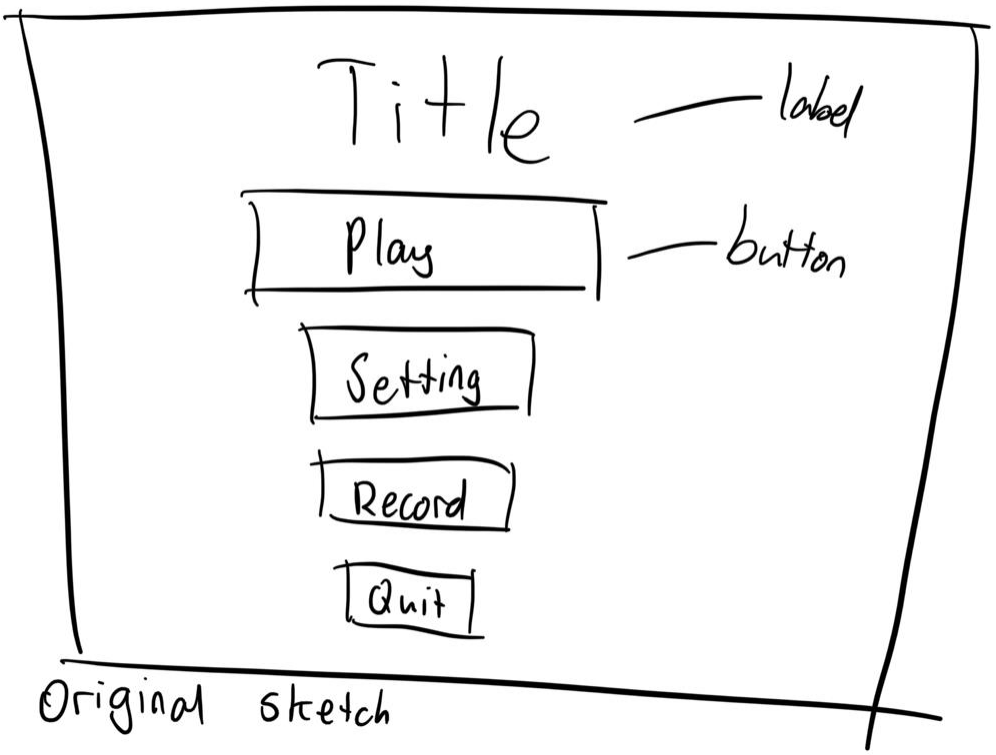
To show the sequence and movement of data and control parameters in the system. The following chart displays the movement of data in relation to a game of golf.

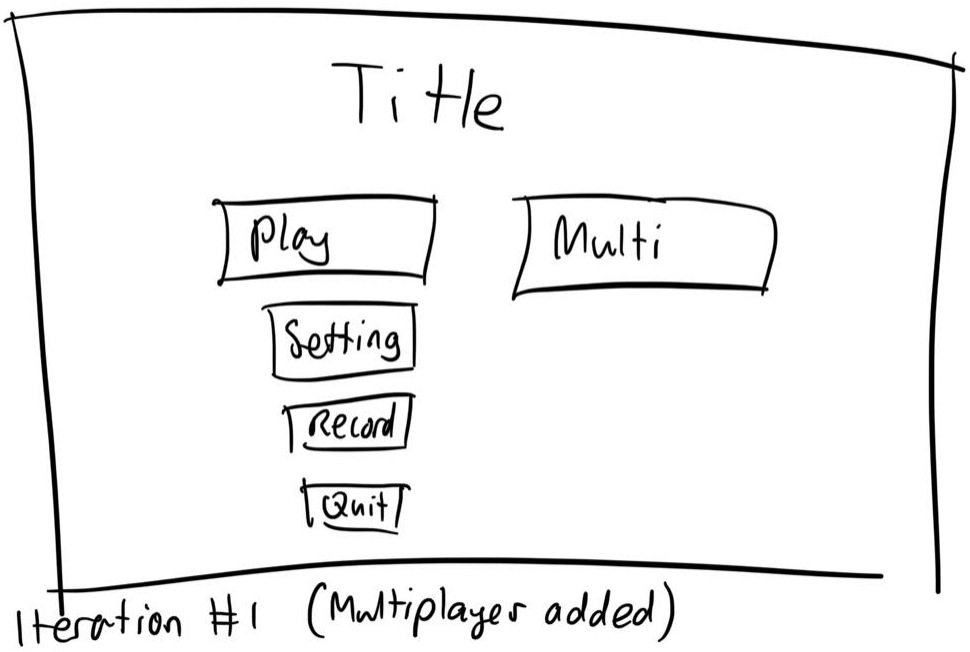


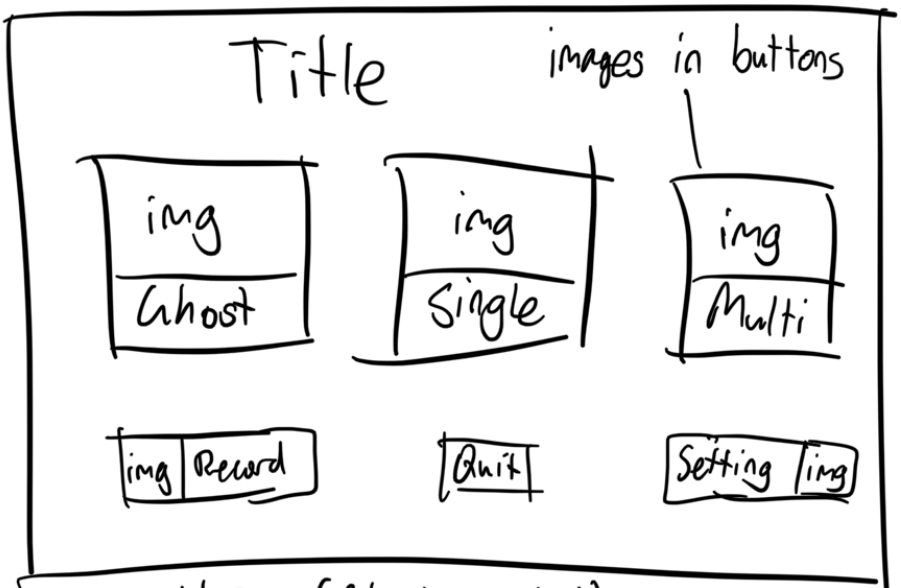
Screen Design

As the prototyping approach is used, multiple UI models were designed before a final one was decided on. The following images outline past prototypes.

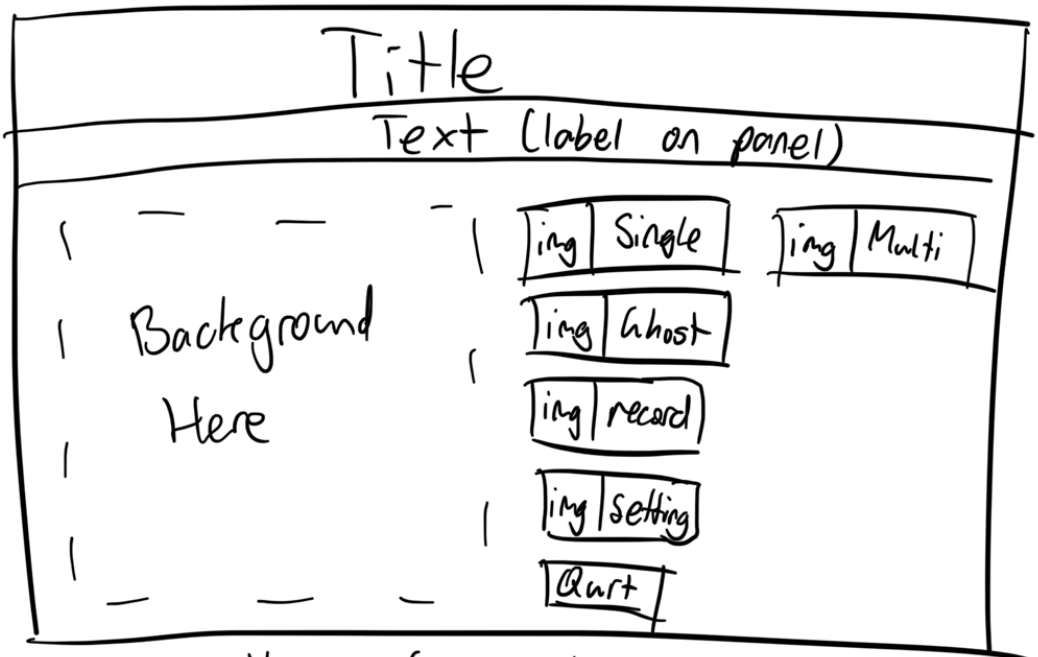
Main Menu Designs





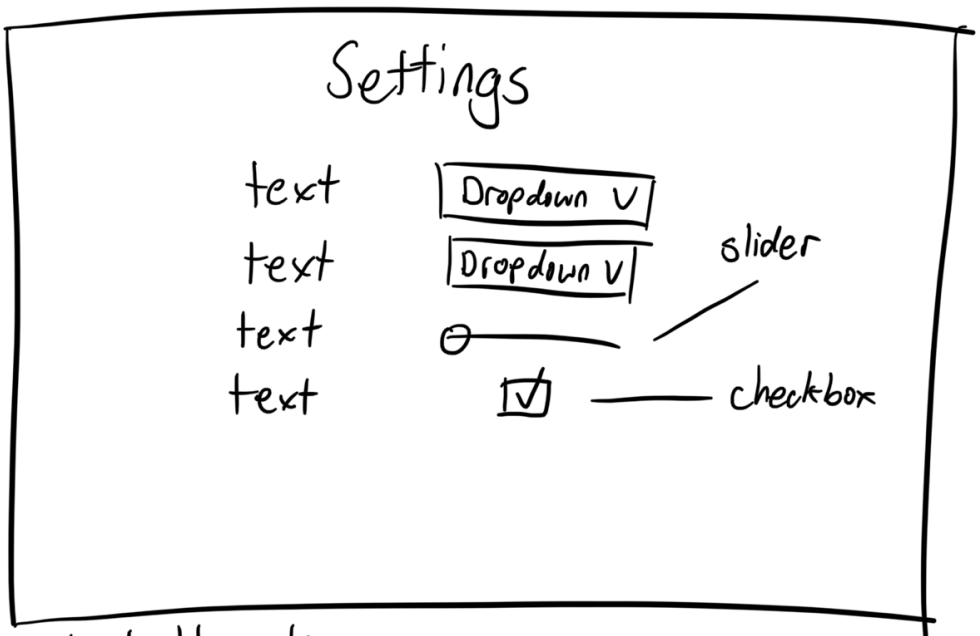


Iteration #2 (Ghost added)

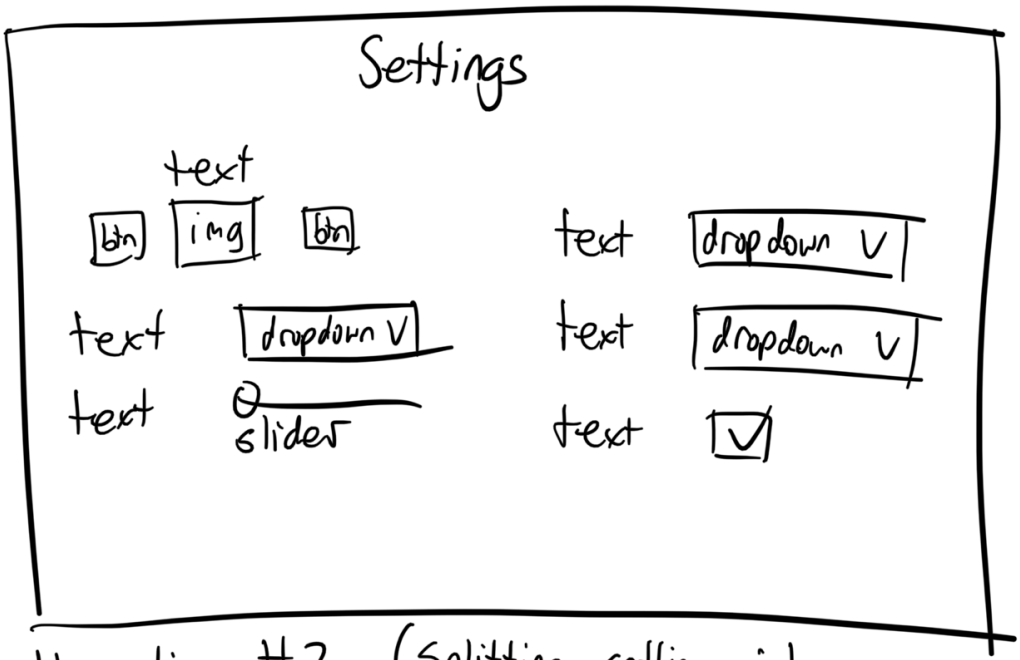


Iteration #3 (Moving buttons around)

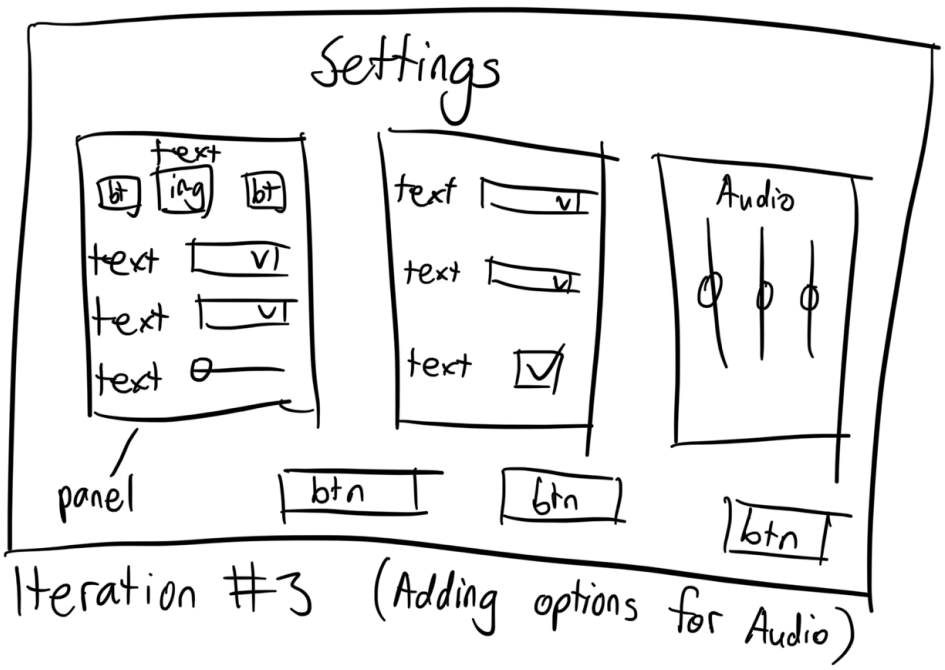
Setting Designs



Initial Iteration



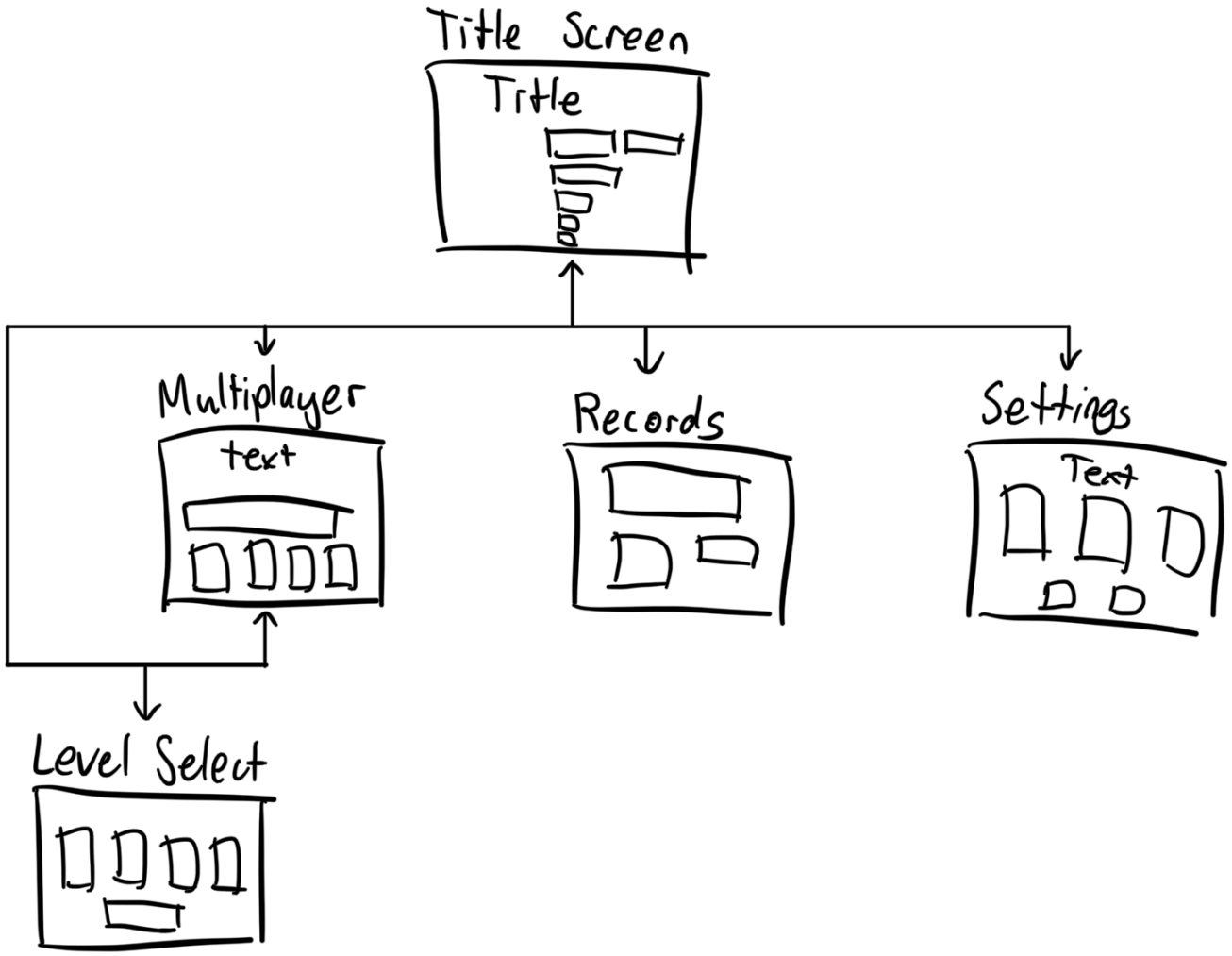
Iteration #2 (Splitting settings into categorical columns)



Iteration #3 (Adding options for Audio)

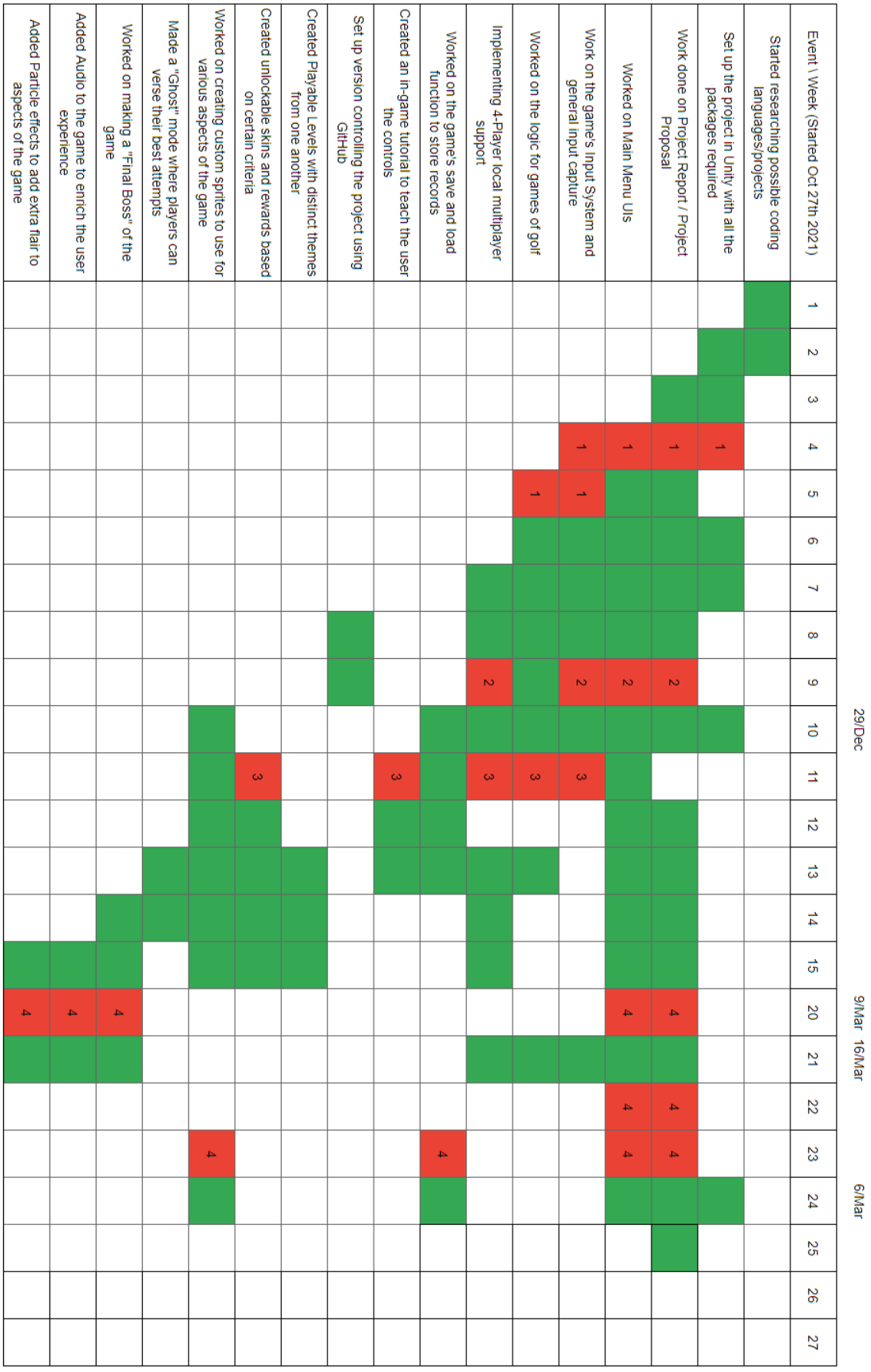
Storyboard

Shows the relationship and the movement between screens. The following storyboard shows the connection between each major menu.



4. Project Management

Gantt Chart



The above chart shows the progress that was made to the solution on a weekly basis. Squares marked in green reveal the weeks when those modules were worked on. Squares marked in red reveal moments when progress was halted due to an issue or event. The following list corresponds to the numbers in the squares:

1. Due to exam preparation, progress on the project was temporarily halted from week 4-5
2. Took time off to enjoy holidays with family
3. Major bug in Input System prevented any progress being made
4. Took time off between Week 16 and 24 due to School and Exam Prep

Project Log

The following are project logs about what progress was made, and what was planned to be done.

Week 0	
To Do:	Done:
Define the problem Plan the solution Research programming languages	
Began working on the project. Need to understand what is being asked of and how to do it.	

Week 1	
To Do:	Done:
Research programming languages Begin working on the project	Define the problem Plan the solution
It has been decided to create an interactive game with the main focus of golf in 2D as it would be simple enough to make within the limited time available. To that effect, the best and most familiar programming environment would most likely be Unity.	

Week 2	
To Do:	Done:
Begin working on the project Start getting all the components together	Research programming languages
After it was decided what language to use, I have started getting all the components required to start to create a game, including installing a version of Unity and Visual Studio that can be used for scripting. It may take a while to get used to the language, C#, as I am not very familiar with it.	

Week 3	
To Do:	Done:
Start getting all the components together Get the basic mechanics working	Started working on the project
After becoming a little more familiar with Unity, I will plan to start working on the actual mechanics of the game. This would include player inputs and the actual “win” mechanics.	

Week 4	
To Do:	Done:
Work on main menu and UIs Update the Input System Set up more packages	
No work was done this week due to starting exam preparation. However after further research, it was decided to update the Input System as it would allow for greater flexibility in terms of accessibility and future-proofing.	

Week 5	
To Do:	Done:
Update the Input System Work on the game system	Work on main menu and UIs
Work was also heavily impacted due to exam preparation. However, some work was able to be done on a prototype of the UI. The Input System still needs to be converted and the game system still needs to be developed.	

Week 6	
To Do:	Done:
Further improve the game system Work on 4-player multiplayer Improve accessibility to the game	Worked on main menu and UIs Improved the golf logic
Work was done in developing the logic for the golf game and tweaking the UI again. An idea was formed to allow for 4-player local multiplayer using multiple controllers. Another idea was to implement an alternate button input method which will greatly improve accessibility to the game.	

Week 7	
To Do:	Done:
Improve multiplayer Overhaul logic Improve input system	Worked on multiplayer Worked on UIs
Initial multiplayer support was implemented. As a result, the logic for the golf game must be overhauled	

to support multiple players. An initial menu should also be added to allow for multiple players joining.

Week 8	
To Do:	Done:
Continue improving multiplayer experience Continue improving UI experience Continue improving input system Implement save system	Worked on multiplayer Worked on golf logic Set up version control software
Continued working on the multiplayer experience and overhauling the golf logic. I should think about getting a save system working so that records and other details can be saved across sessions. Version control was also set up to keep regular backups of the project data and a better track of day-by-day progress.	

Week 9	
To Do:	Done:
Improve UIs Improve input system Improve multiplayer	Logic for golf Finishing setting up version control
Little work was done so that I could spend time with my family for the holidays. However, some work was done in improving the logic for the game of golf. And version control was finished being set up.	

Week 10	
To Do:	Done:
Create unlockable skins Create a tutorial Keep working on save system	Started making custom sprites Started making a save and load system Continued working on input, multiplayer, and UI
Custom sprites need to be made so that unlockable skins can be implemented. As such, a save and load is also implemented to be able to store the user's best scores and unlocked skins. There is currently a plan to create an interactable tutorial with some dialogue.	

Week 11	
To Do:	Done:

Update save system with new data to save Make a tutorial Keep working on unlockable skins	Fixing input system
Due to a major bug found within the input system, work for this week was almost entirely halted until this issue could be fixed.	

Week 12	
To Do:	Done:
Tweak golf logic to support skins Create rest of playable stages Keep creating new sprites Add "Ghost" feature	Started making a tutorial Started making unlockable system
Input system was fixed so progress could continue. A tutorial was made that would interactively guide the user through a game of golf. An unlockable system was also implemented to reward players with skins the more they played. As such, the game's logic must be tweaked to support these changes.	

Week 13	
To Do:	Done:
Create Levels 3 and 4 Start planning a "Final Boss"	Tweaked golf logic Added Levels 1 and 2 Preliminary Ghost added Finalised the tutorial
The tutorial was finalised and so the rest of the levels could start to be implemented. Thanks to a template of the levels, modules and sprites could be reused which greatly saved time and resources. The rest of the levels need to be implemented as well as a "Final Boss" that is planned to use totally different mechanics to the main game.	

Week 14	
To Do:	Done:
Add multiplayer support to "Final Boss" Add audio Add particles Update save system	Finalised Ghost mode Added Levels 3 and 4 Started developing "Final Boss" mode
Finished adding the last few levels and finalised the Ghost mode. As well as started developing the "Final Boss" which has been decided to use "states" as different phases and animated keyframes for movement (a new endeavour for me in Unity). Adding audio and particles would also be a good way to more familiarise myself with different parts of Unity.	

Week 15	
To Do:	Done:
Improve overall UI Finalise audio and particles	Added audio support Added particle effects Finalised Final Boss phases Updated main menu UIs
Final Boss was finalised with 5 distinct phases and different attack patterns. Audio in the form of background music, UI effects, and In-game effects were also added. As well as particles in certain places. Now to improve the overall UI as well as finalise audio and particles.	

Weeks 16-23	
To Do:	Done:
Improve UI Finalise audio and particles	
Due to the beginning of the new school term, progress was essentially halted between these periods. However, there are still plans to improve the UI through further prototypes.	

Week 24	
To Do:	Done:
Final polish of the software Re-make loading screen	Updated security of save file Revamped background of the main menu
The background to the main menu was entirely redone in 3D to take advantage of modelling tools in Unity. External programs were also used to create models for this background. Additionally, a checksum was added to save files to ensure that data isn't majorly tampered with or corrupted upon saving.	

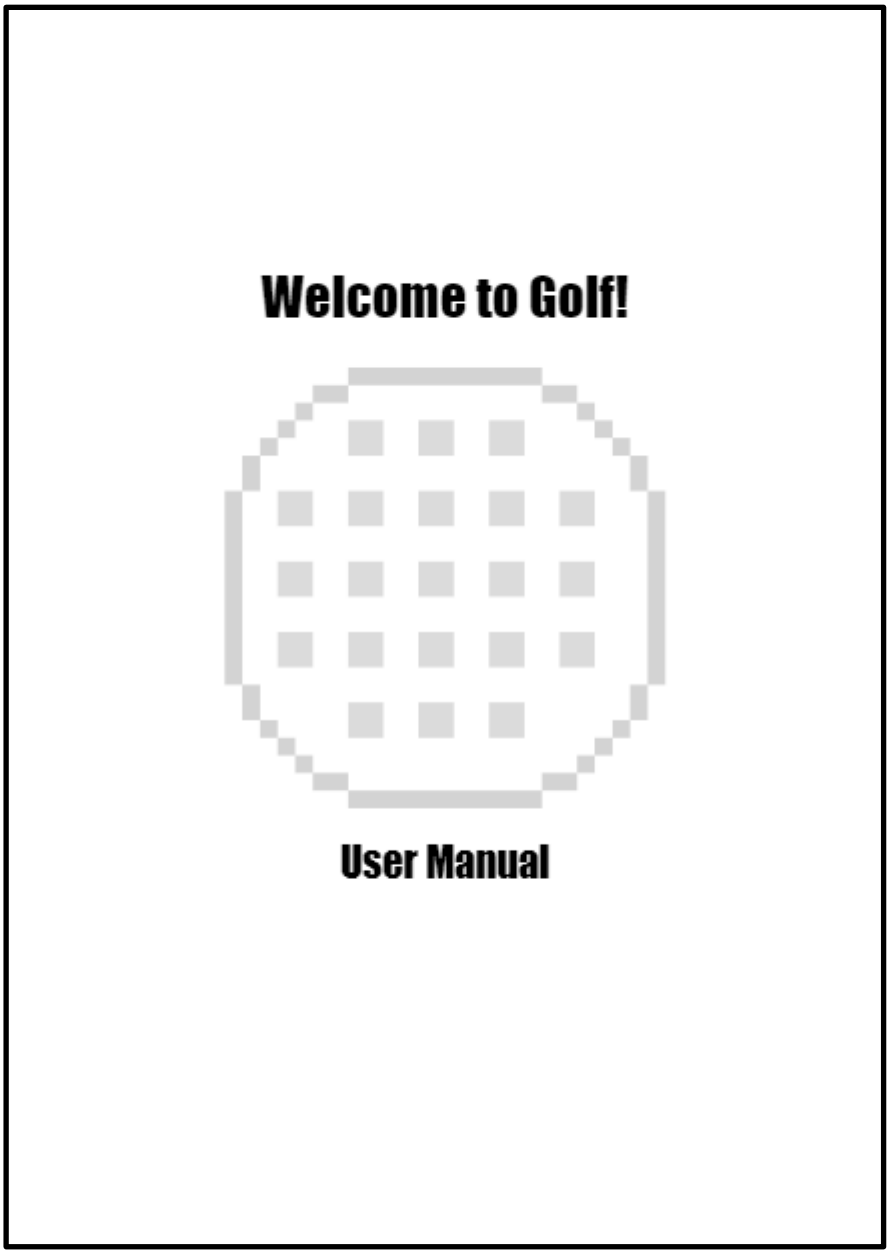
Week 25	
To Do:	Done:

Keep polishing the software Working on documentation	Re-made loading screen
As the bulk of the software has already been completed, all that's left to do is polish the game, fix bugs, and clean up code. The loading screen was updated to look more appealing and to support reduced motion. Technical specifications and screenshots need to be taken to be added to the documentation for the game.	

5. User Documentation

Please see the document titled “User Manual” for user documentation.

Included is an overview of the Main Menus, the modes, controls, and troubleshooting/FAQ.



6. Developer Documentation

Design and Function

The program was designed so that it is both visually appealing and impressive, but also inclusive and functional across multiple input types. As a result, extreme care was taken in making sure the User Interface was intuitive to use when using either a mouse, keyboard, or controller.

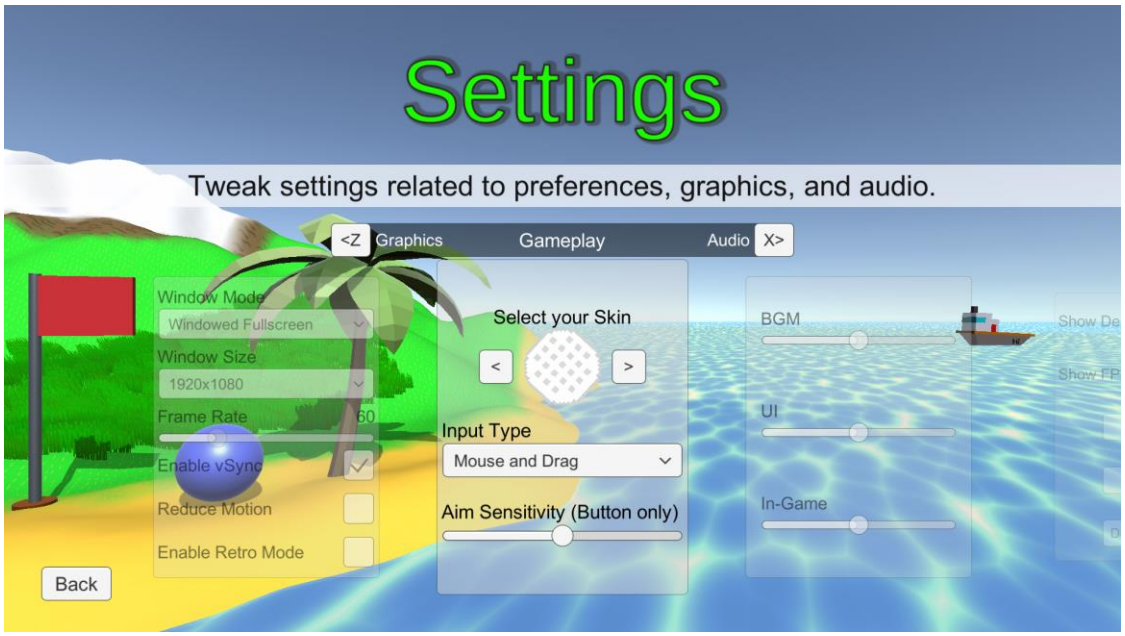
Main Menus

This design philosophy can be found the first moment the user opens the application and is met with the Main Menu. Vector lines from top to bottom have been utilised to show the most important items at the top and going down to the lesser important items down the bottom.

Both Singleplayer and Vs. Ghost are modes intended for solo play and beating records. While Multiplayer is intended for playing against friends or family where up to 4 players are able to play at the same time. Beneath those are Records where users can view their achievements and unlocked items. Settings is where users are able to customise the application. And then Quit where the user can close the game. The Hide UI button lets the user hide the UI to view the custom designed 3D background fully.

In between the title and the buttons is a banner that reveals a small description of any item that is being selected by the user. Its function is almost identical to a balloon tooltip and aims to assist the user in navigating the software.

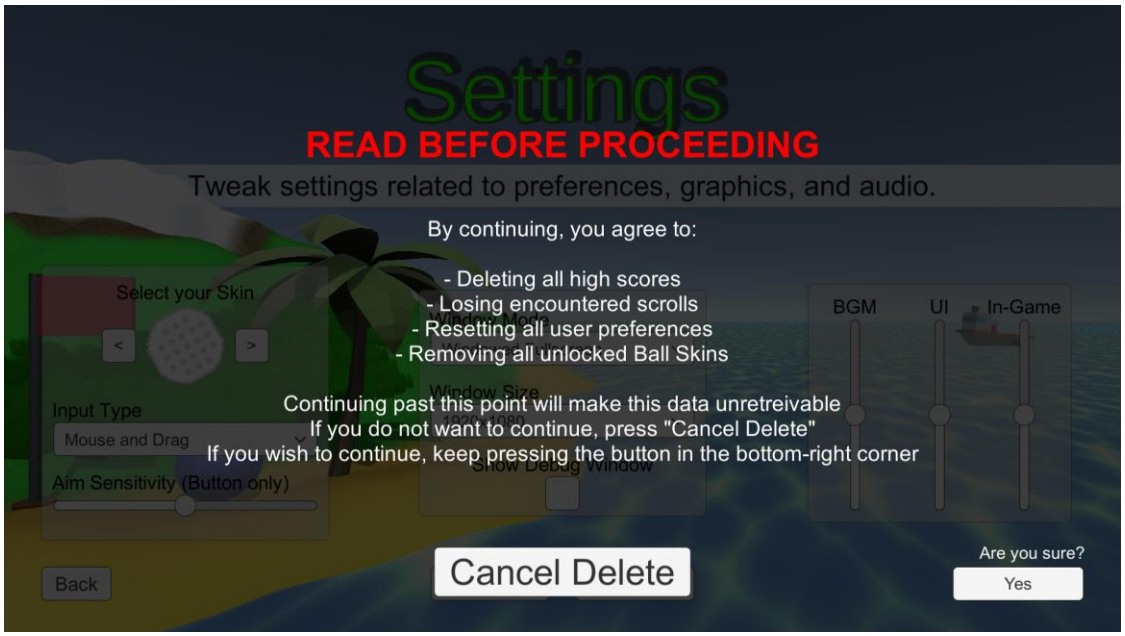
To the left of the buttons is a small panel that informs the user about progress that is made on the game with every release. This will allow the user to feel more involved with the development of the software. The text is decided based on a text file that is downloaded from the internet.



The Settings menu has been crafted so that each option is grouped into similar groups. This allows the user to quickly and easily identify the purpose of different settings. In addition to that, the banner in a similar location to the Main Menu will also give a brief description about any items that are being selected.

In particular, the options are sorted into “Graphics”, “Gameplay”, “Audio”, and “Other”. Using the intuitive buttons or using the corresponding keys, the user can transition between each panel seamlessly to access similar settings.

Additionally, the user is given an option to Delete All Data if they wish to. Because this is a major and irreversible action, the user is given forgiveness as a warning screen is brought up before their data is actually deleted, giving them a chance to cancel before irreversible damage is done.



To allow for greater inclusivity, there are options to either use mouse controls or keyboard controls. This way, users who are unable to perform complex actions with the mouse will still be able to interface with the game in different but equal ways.

The game also supports a range of resolutions to support a variety of screen sizes. While most of the resolutions target the aspect ratio of 16:9, there is an option for a 4:3 aspect ratio. This was a heavy consideration when designing the software as all menus and screens must work on both aspect ratios correctly.

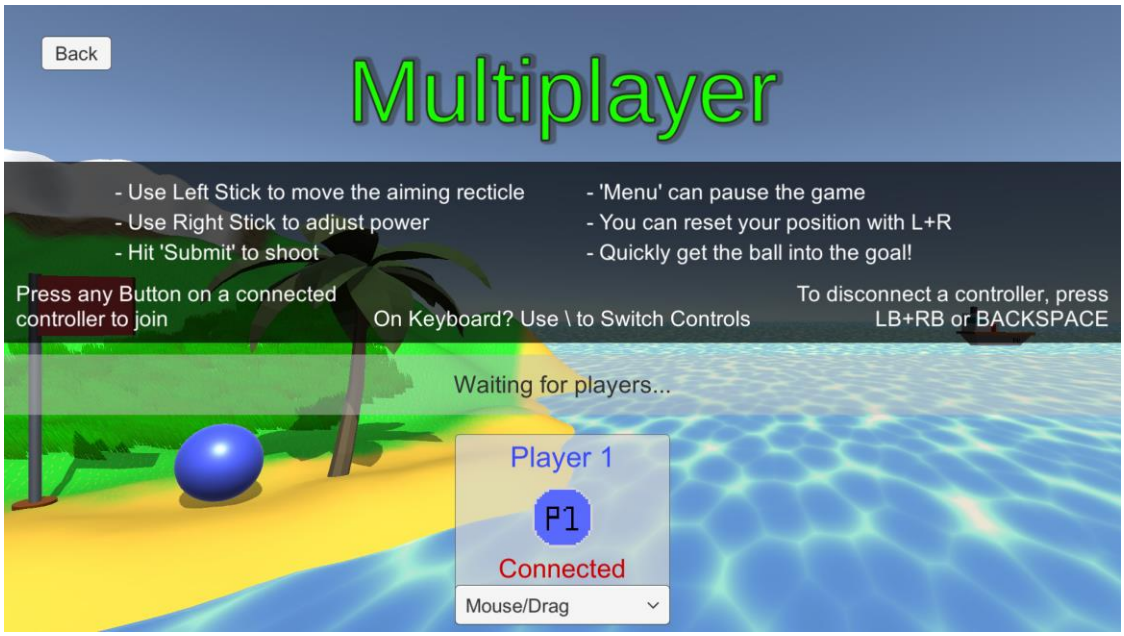
The Debug Window toggle is a slightly more advanced option as it reveals output messages from the game. While output messages are usually only meant for the developer as a method of debugging, I thought it would be a nice addition to let more advanced users view the processes that are occurring in the background of the game.

Advanced users would also appreciate the built-in FPS Counter, which displays the number of frames the game is producing per second. Combined with the Frame Rate slider to adjust the game's framerate and the vSync toggle to prevent screen tearing, advanced users will have the ability to customise the game's performance to their liking.

Reduce Motion was an option specifically added for accessibility and to promote greater inclusivity. As fast and rapid motion may cause pain and discomfort for some users, this toggle would either disable or reduce the speed of any quick moving objects.

The Records menu also offers a clean and convenient way for the user to view their achievements, statistics, and unlocked items. Groups have also been laid out logically to allow the user to see all of their unlocked items at a quick glance. Similar to both the Settings and Main Menu, a banner gives a quick description about each item.

In order to add extra depth and enjoyment to the game, the Ancient Scrolls can be selected to bring up a short story related to the stage they were found in. Level Records also reveal the user's best records in a course which can be toggled through with the buttons to move to the next or previous stage.



If the user chooses to play a Multiplayer game, they will be sent to the Multiplayer menu first so that additional players will be able to join. As additional players may not have prior knowledge of the game, a brief explanation on the controls and goal of the game can be found right in the centre of the screen. Below that is the button to begin once all players have joined. The text inside the button will indicate whether enough players have joined or not. And the button will no longer be greyed out, as is standard for UI elements that cannot be selected.

When players join the game, a player card will appear on the bottom of the screen. Due to limitations with Unity, only 1 player can use Mouse or Keyboard at a time. However, that player is given the option between the two different Input Types that is normally chosen from Settings.

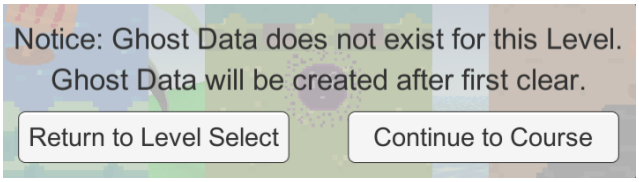
For all other players, they must join through a compatible Controller and therefore must use the ‘Controller and Buttons’ Input Type.

Once all other players have joined, Player 1 will then be able to press Play to go to Level Select. As Player 1 is considered the “host” from now, only they will be able to navigate the menus until this “session” is disbanded by returning to the Main Menu. Note: Player 1 is not necessarily always the Mouse/Keyboard player. It is just the user who joins the game first.

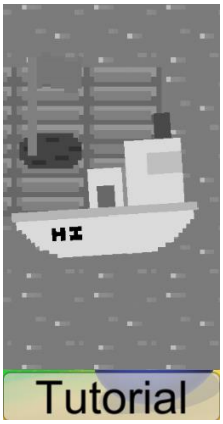
If the user selects Singleplayer or Vs. Ghost instead, they will be taken straight to the Level Select screen to choose the course they want to play on. This is also the screen that directly follows the Multiplayer screen after the Play button is pressed. Under the title is also a banner that tells the user their current mode instead to avoid ambiguity.

Each of the different stages shows a small snapshot of the course to give the user a visual teaser before they enter it.

In Vs. Ghost, if the player attempts to enter a course that does not have a previously set record, then they are asked whether they still want to continue.



Due to the Dialogue System used in the Tutorial, it is unavailable in Multiplayer. To indicate this, the icon for the Tutorial level is greyed out, as is standard for most UI and thus consistent with the Operating System too.



In a Game of Golf

Once inside a course, the Head Up Display (HUD) outlines vital information to the user during the game. The above image labels the essentials of every course.

4 is the timer that shows the amount of time that has passed. As speed is vital in this game, it is imperative that the timer is in an easily accessible location. Next to the timer is also the Scroll indicator, which shows whether the Ancient Scroll has been collected or not. This is shown through whether the scroll is greyed out or not.

5 is also important as it is the player indicator. It includes dedicated information for the user. In particular, it shows how many shots they have taken and their final time once they reach the goal. In Multiplayer, multiple player indicators will be displayed on the corners of the screen.

In order to maintain constancy, the same HUD is utilised across the other golf stages. That way, the user can learn the game with the Tutorial, and then apply the same skills across the rest of the game, reducing the amount of time having to learn the game. This was also done to increase the amount of resources that could be reused as a part of the RAD approach.

1, 2, and 3 are also present in all other stages. 1 is the player's ball and aiming indicator. By performing the required inputs, they will be able to aim the ball, set the power, then shoot the ball. These actions are determined by the Input Type the user is using and described in "*Getting Input for Golf*"

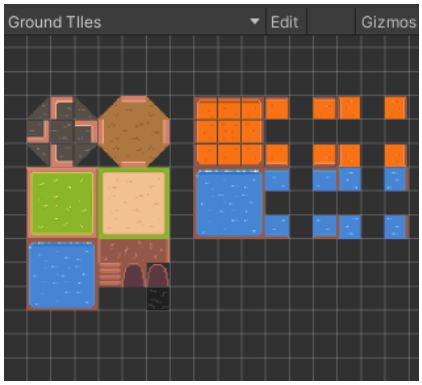
2 is an Ancient Scroll, the collectables in these stages. These have been added to give the user a sense of achievement in finding something, and then being rewarded for it in Records.

3 is the goal. Reaching the goal as fast as possible is the player's objective. The goal has been specifically selected as a flag as it represents the real life game of golf, and thus the player will be more familiar with the goal of the game.

As a part of the RAD approach to be resource and time efficient, tilemaps were utilised to construct every golf level. Tilemaps consist of a grid of squares which are filled in by tiles.

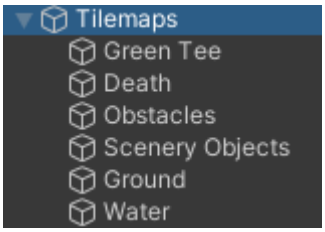
Tiles allow for easy and effective painting of background and foreground elements that are required for a 2D game. As tiles are based on sprites, they are easily sourced from the Unity Asset Store, as well as modified to meet my own needs, while respecting the original creator's Intellectual Property rights.

Tiles are chosen from a 'Tile Palette' which groups similar tiles together. For example, 'Ground Tiles' groups tiles that are used as ground elements. And 'Objects' are used for tiles that are meant to be used as either decoration or as an obstacle.



Due to limitations with the built-in tile system, an external Package called '2D Tilemap Extras' allows for animated tiles to be added and to allow ground elements such as water and lava to feel more alive.

Multiple tilemaps can exist in a scene at any given time, and this is used to layer different objects on top of each other. Such as flowers and trees on top of the ground. And having the ground be on top of the water. This method of designing levels allowed for quick development of each stage as only new tiles were required for each new theme. Any other method would have taken considerably longer and with a higher resource cost.



Tilemaps are not used for the player's ball nor the flag. Those are just 2D sprites that have been placed in the scene over the tilemaps.

If the player needs to, they are able to access a Pause Menu during the game. This Pause Menu is also consistent across all other stages to promote easy learning of the game. The menu shows the player that paused the game, buttons to Resume, Restart, or Return to Main Menu, as well as the controls for the player's Input Type.

As there are multiple players in Multiplayer, it is important that all players know who paused the game. Additionally, only the player that paused the game will be able to navigate the Pause Menu, as is standard with other common multiplayer games.

The controls text was also an important decision as it would quickly allow players to check the controls without having to look through the User Manual, and would help to streamline the process of learning the game.

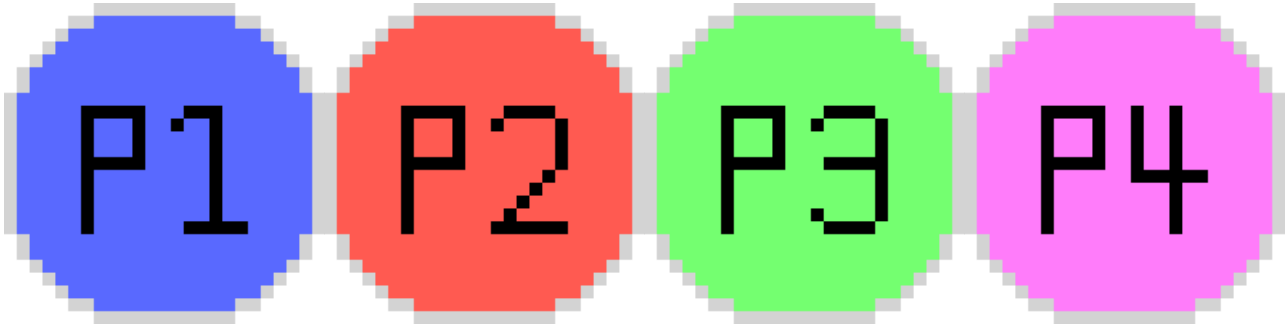
Once all players have reached the goal, the Course Clear screen will appear.

In Singleplayer and Vs. Ghost, it will tell the player the time it took for them to clear the course as well as how many shots it took. Then it will say whether they have beaten the record or not. This text is used as inspiration for the player to set new records, and is a crucial factor in the replayability of the game.

In Multiplayer, the screen will show the player who was able to reach the goal the fastest, as well as inspiring the player who may have performed the worst to try better next time.

Below the text, the options to Restart Course, Return to Level Select, or Quit to Main Menu can be selected. These buttons were specifically worded to avoid ambiguity between the options and to make the functions of these buttons clear.

Just like the other in-game UIs, this screen is also reused across the other levels so that the user will always know that the same buttons will lead to the same place.



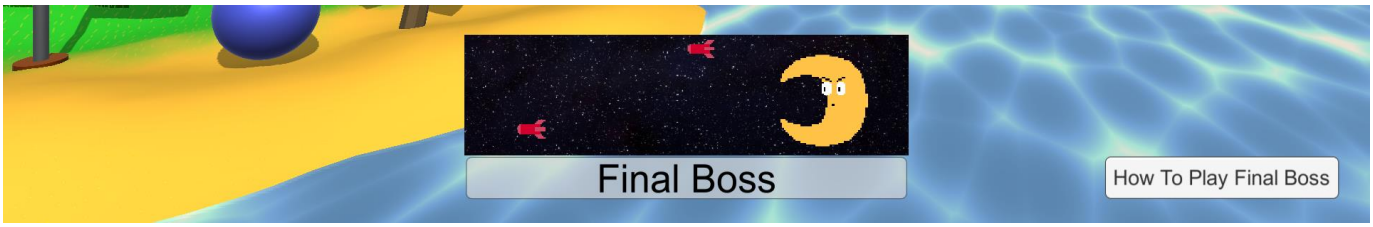
As seen with the above images, a Singleplayer game will result in a standard ball for the player to control. This standard ball can have a skin applied to it from the Settings menu and more skins can be unlocked as the user plays through the game.

However, as there are multiple players in a Multiplayer game, custom balls had to be made to avoid ambiguity between different players. As such, each ball is indicated with the number of the player. As well as a different colour per ball to make it even easier to identify who is who.

These colours also correspond to the player's indicator on the HUD so if they forget which player or which ball they are, they will be able to quickly identify themselves in the heat of a match.

In the Final Boss

The Final Boss is a bonus mode that was added for players who have cleared all other regular levels. It aimed to add a different experience on top of the regular golf game and to be able to utilise different aspects of Unity.

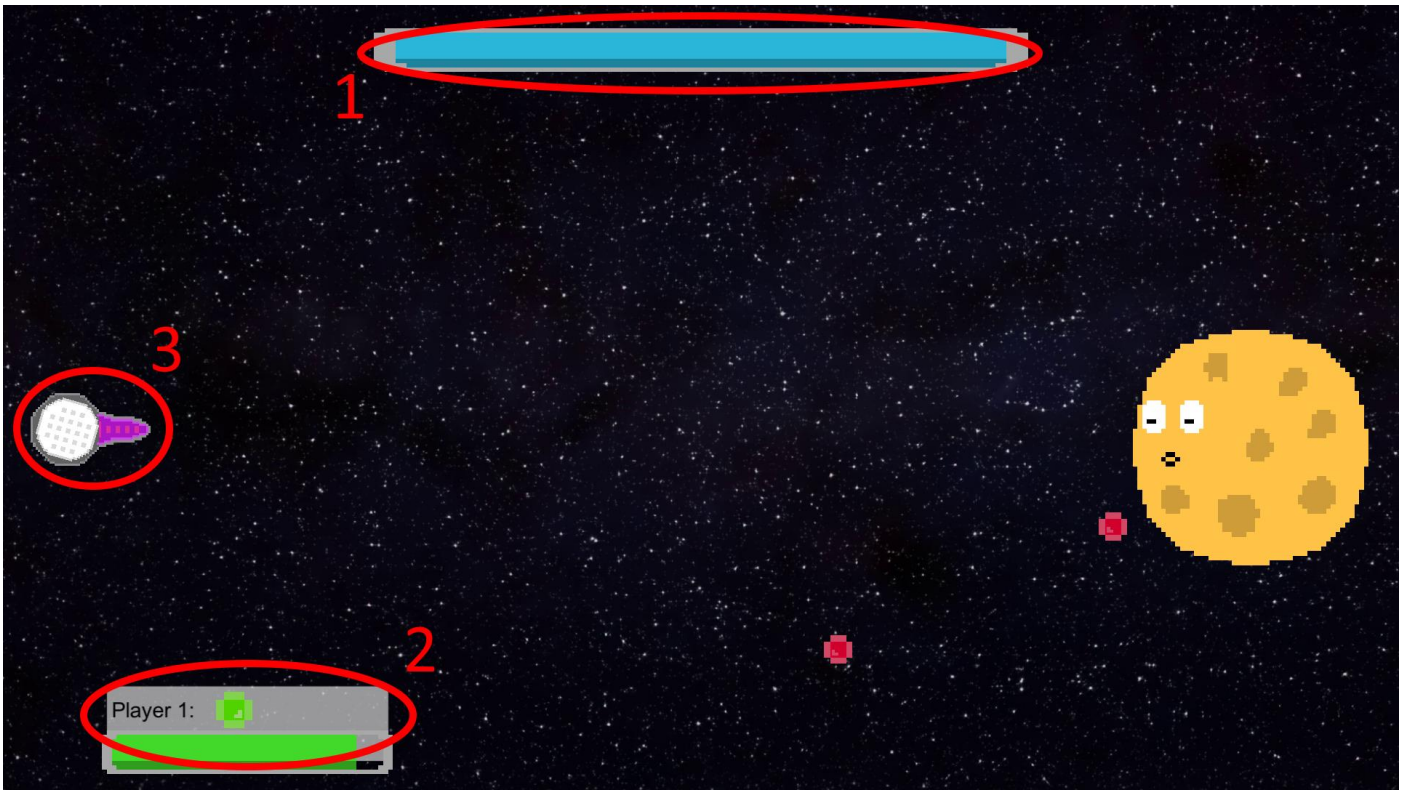


As the Final Boss is based on random attack patterns and completely different inputs, it is unavailable in Vs. Ghost and its icon becomes greyed out in that mode, as standard with other icons.

The Final Boss is based on an on-rails shooter where the aim is to deplete the Boss' Hit Points (HP) as fast as possible. This mode also supports the multiple Input Types available in the other levels for inclusivity.

The Boss attacks in five phases, with each phase becoming more difficult as its HP decreases. The player has the option of 2 attacks which allow for a greater range of depth and strategising to beat the Boss as fast as possible.

While the Pause Menu and Results Screen remains virtually identical to a regular golf level, the HUD is modified to better fit this specific mode.

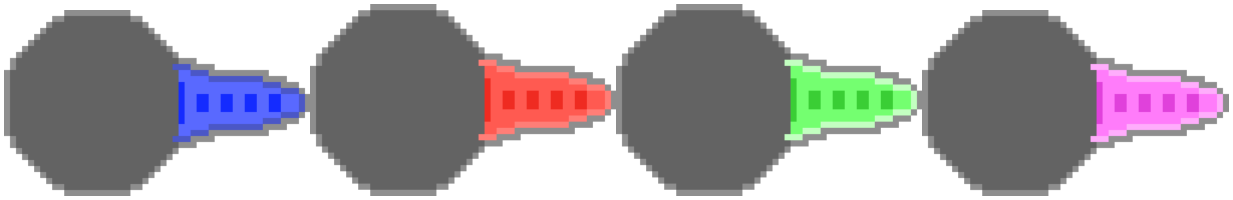


At the top of the screen, replacing the timer is the Boss' Health Bar (1). As the goal is to deplete it as fast as possible, it makes sense that it would be one of the most dominant features on the HUD.

Instead of being anchored to the corners of the screen, the player indicators (2) are now along the bottom of the screen. And in a Multiplayer game, they will be lined up along the bottom of the screen instead. This allows for more empty space to be retained which will let the player view more of the action.

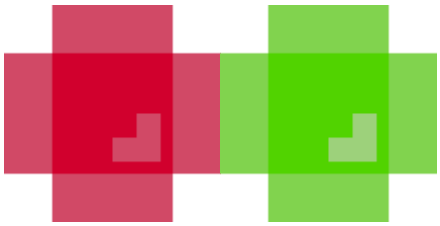
The player indicators similarly have the player's number but also their current projectile, the weapon's cooldown, as well as the player's health. At a quick and convenient glance, all this information can be gathered which is vital for a fast paced mode such as this.

Just like a regular Multiplayer match, each player's ball and ship (3) will get a different look depending on their player number to avoid ambiguity.

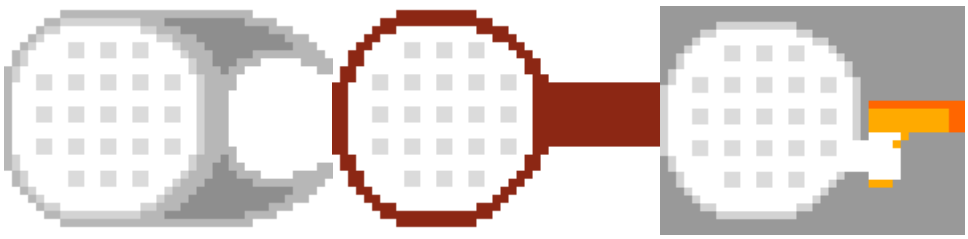


If a player gets hit by the Boss' attack, their ship will begin flashing for a small period of time. This is a visual indicator to the player that they have been hit, but also to indicate their short period of invulnerability where other attacks will not damage them during this time. This is consistent with other action games where flashing characters indicate invulnerability.

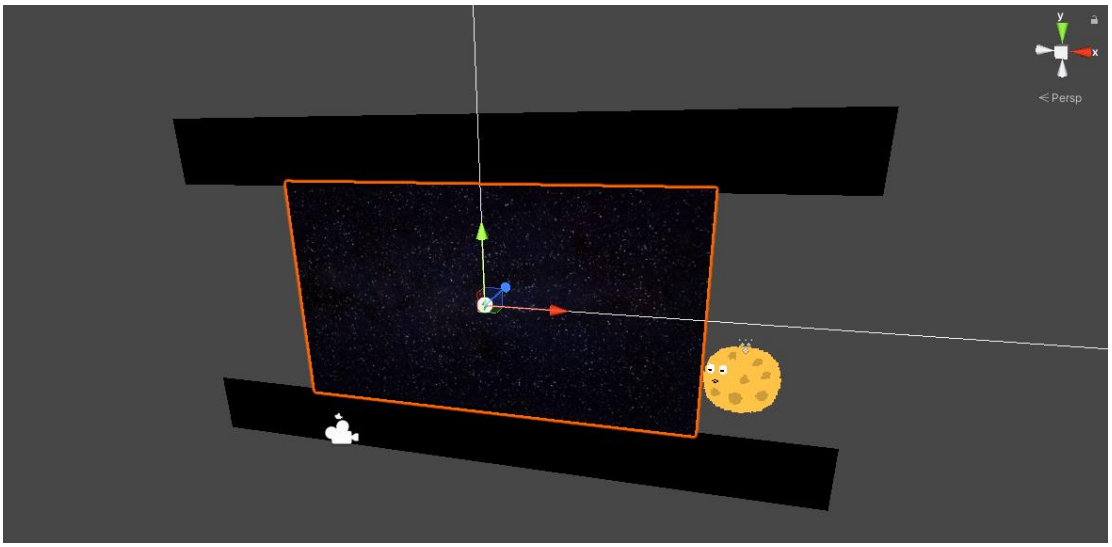
The colours of the projectiles have also been specifically chosen to be easily identifiable for the players. The Boss' projectiles appear as red as it is generally the colour of "evil" and "danger". Meanwhile, the player's projectiles are green as it symbolises "safety" and "goodness".



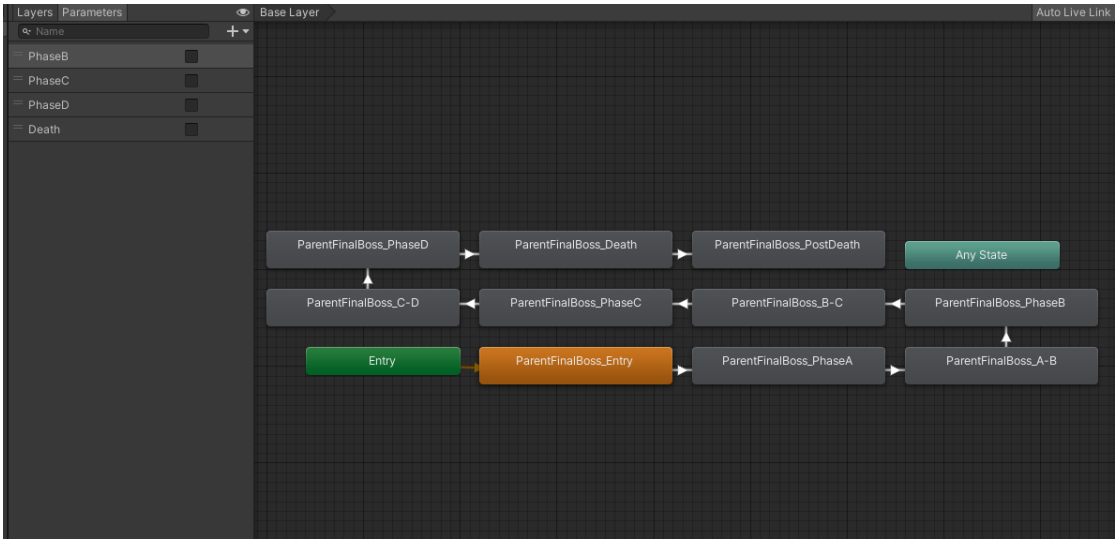
During development of the software, a number of prototypes for the Player's Ship were designed. Ultimately, the final design was chosen as it was both stylish and fit the target style.



Unlike a game of golf, the Final Boss does not use tilemaps. Instead, the background is a quad plane with an image attached to it. The camera does not actually move either. A script is used to scroll the background to give the illusion of flying through space. This was a much more efficient and easier method than having the players and boss move towards the right with the camera following them (which would have been quite computationally taxing).



The Final Boss was also specifically designed to take advantage of Unity's 'state machines'. "States" allow different animations to play at specific times, such as when a certain action takes place or if it's triggered by a specific criteria. In this case, "States" allow us to track different attack phases for the Boss, as well as to play different animations for each phase.

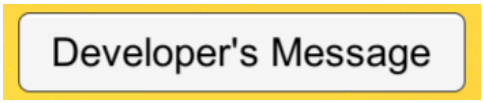


Booleans are used as flags so that it is known what phase the boss is currently in. Toggling these flags tell Unity when to progress to the next "phase" of the battle. Once all phases have been cleared, it will be known that the Boss has been defeated and to end the battle.

To see the pieces of code that handles the Final Boss, please see *'Behind a Final Boss'*.

In the Secret Enemy Rush

The Secret Enemy Rush was another secret mode added to the game to truly commend players for clearing all stages and collecting all Ancient Scrolls. The aim was to widen the scope of the game while providing a suitable reward to the player for completing the game. Unlike the Final Boss mode, the Secret Enemy Rush is accessed by either holding ‘Q’ on a keyboard or ‘LB’ on a controller while entering the ‘Developer’s Message’ menu.

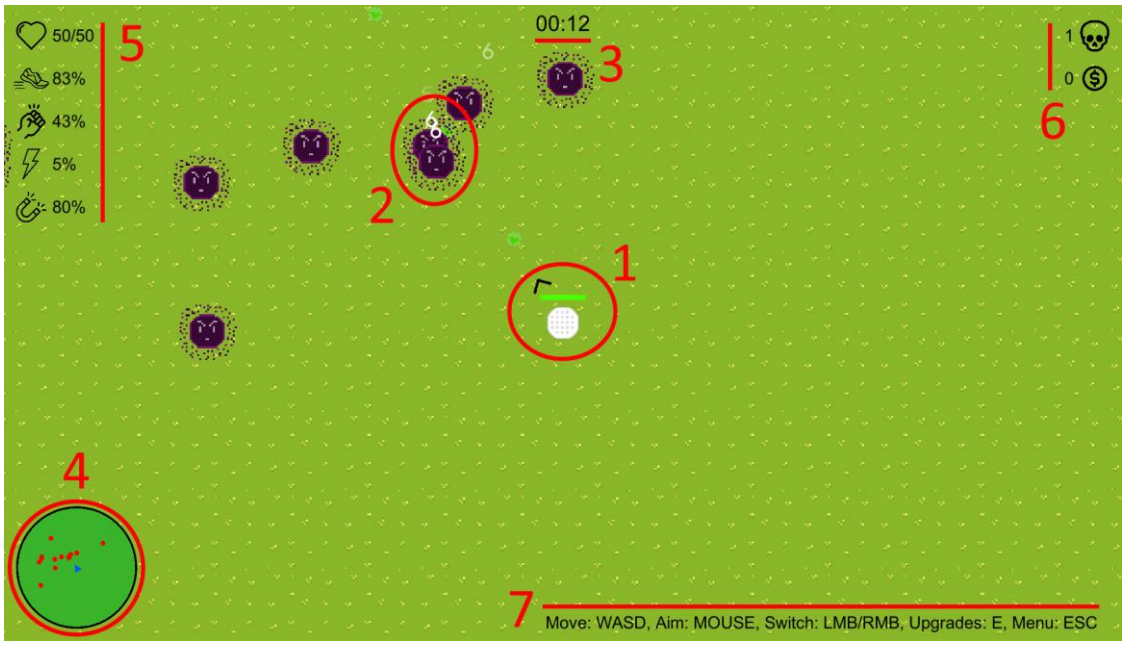


This mode is based on a traditional top-down shooter with the aim of clearing out hordes of enemies within 30 minutes. With each enemy cleared, there is a chance that it will drop ‘Cash’. This cash can be used to purchase upgrades to the player or to their weapon. These upgrades include the ability to increase the Player HP, Player Speed, Weapon Damage, Weapon Crit Rate, and Pickup Range.

Progressively throughout the game, more and more enemies will be spawned per group. This ensures that the game is scaled properly according to the upgrades that the player is expected to have by that time. When the game reaches 30 minutes, the player is considered to have beaten the mode. Enemies are also spawned in a random location just off-screen.

The player also has a choice between 2 firing modes. A faster mode that shoots in the direction they are aiming, or a slower mode that shoots in 4 directions around them. This allows the player to choose their choice of playstyle.

Thanks to improvements in my own abilities with coding in Unity, this mode does not follow the Input Type from Settings. Instead, the game will intuitively switch input types whenever the user switches their input device. Therefore, the user can seamlessly switch input devices without having to pause or exit the mode.



Because of the widely different game style, a custom HUD was designed for this mode. The player (1) is easily identified in the centre of the screen with their health bar slightly above and an arrow indicating their current direction of aim.

The enemy (2) shares the same sprite as the enemies from the Golf mode as a resource saving measure and to promote consistency across the entire game.

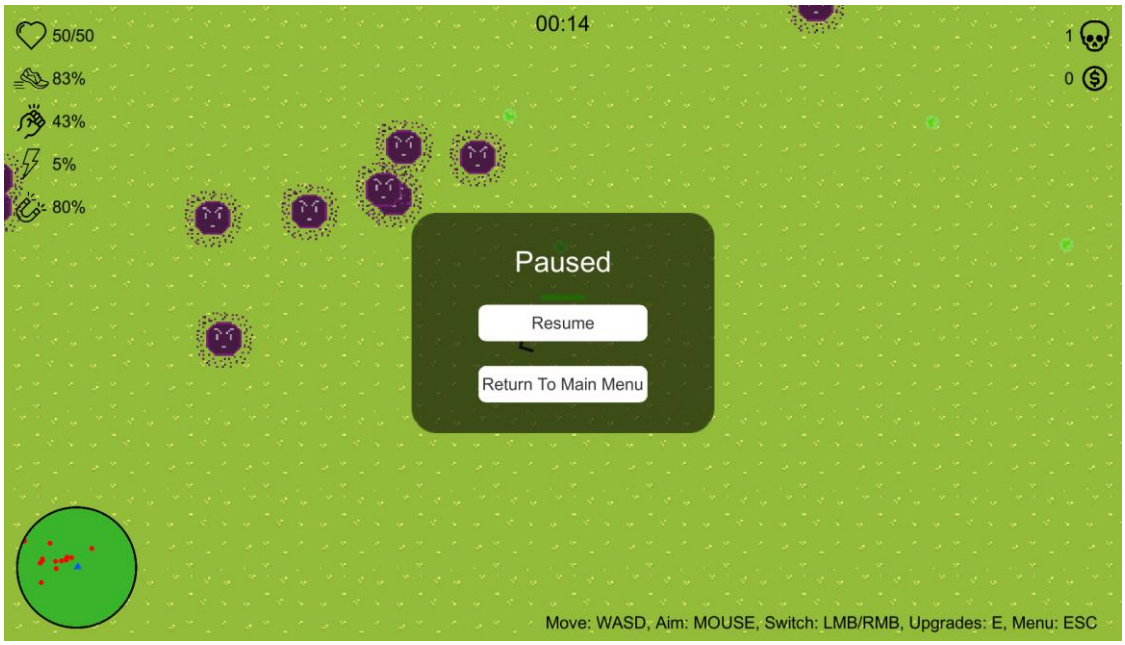
Because of the time based goal, the timer (3) is visible on the top of the screen. This allows the user to be able to quickly and easily see how long they have lasted thus far.

The unique edition to this mode is the minimap (4). To prevent the user from being caught off guard and to more easily see where coins are located, the minimap follows the user's current location and displays any relevant entities in the area.

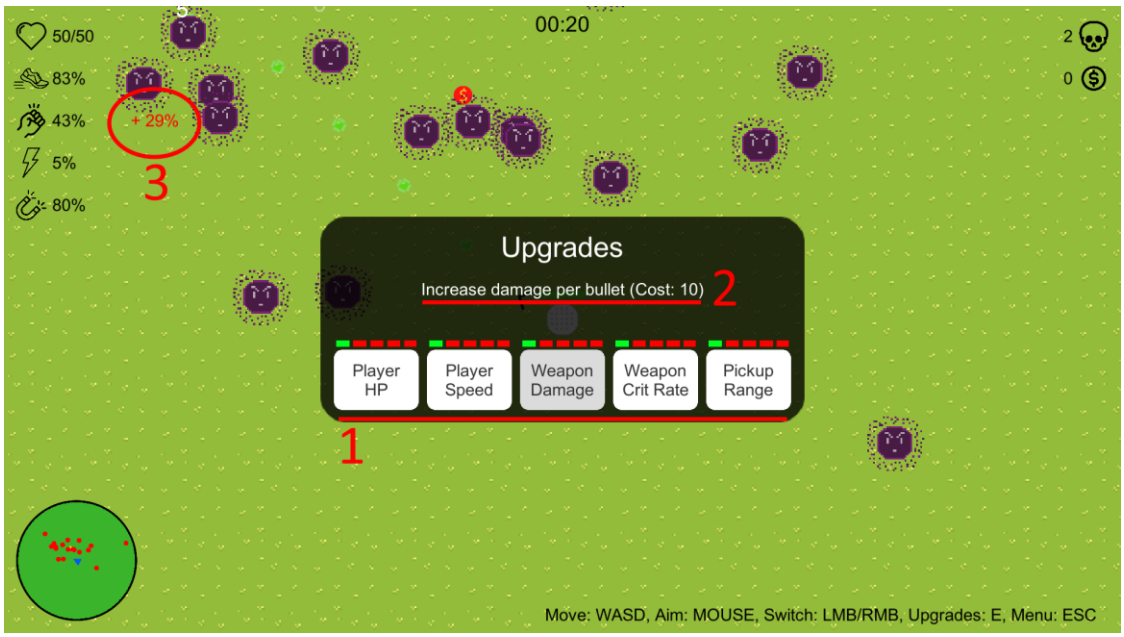
Upgrades (5) are displayed on the top left corner of the screen. This easily allows the user to see their current progress and provides a numerical value for their health remaining. The list also matches the order of the items in the Upgrades menu.

Other statistics (6) are displayed on the top right of the screen. This shows how many enemies have been defeated, plus how many coins the user currently has. When purchasing an upgrade, the required amount of coins are automatically deducted from this amount.

Since it is more convenient, the user's controls (7) are displayed along the bottom of the screen. Because this is also a secret mode, the instructions are not available in any other part of the game. The only mention is in the "User Manual". These instructions automatically update if the user decides to change input devices during the game.



When the game is paused, a slightly different pause menu is used which fits the theme of this mode. The only two options are to Resume or Return to Main Menu.



Since upgrades are a big part of this mode, an Upgrades menu has been added to facilitate this function. Each of the upgrades (1) are listed in buttons which can be selected to confirm the upgrade. Above each button are little green/red indicators to show the upgrade's current level. Consistent with colour theory, green indicates the user already has that upgrade. While red indicates the user has yet to unlock that upgrade.

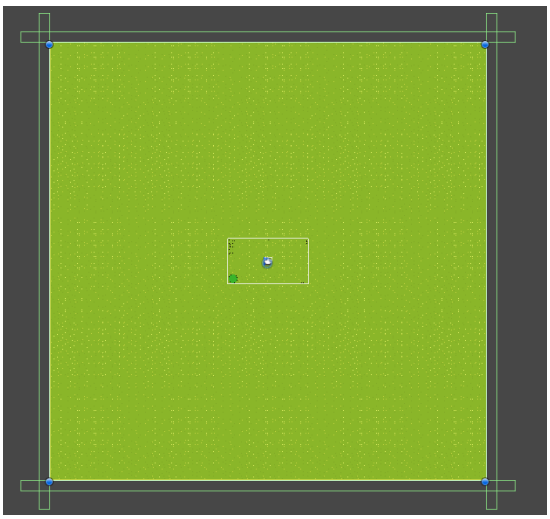
To provide more information about each upgrade, descriptor text (2) shows that the upgrade is, along with how much it will cost. This text changes depending on what upgrade the user is currently hovering on.

For more specific information on the upgrade, upgrade text (3) shows a value which will be added onto your current statistics if the upgrade is purchased. This allows players to make more informed choices on what upgrade to purchase during the game.

Unlike regular pause menus which use one button press to open the menu, then another button press to close it, the Upgrade menu requires the user to hold a button to keep it open. This allows the user to more seamlessly move between game and menu actions.

The Upgrade menu will slow down the game to a fraction of its full speed. This allows the user to "feel cool" as they select an upgrade. While the menu is open, the user does not have control over their player, nor their aim. As a result, the player character will continue moving in the same direction as before the menu was opened. The character's aim will also be locked to its current direction. This will prevent the player from quickly being swarmed by enemies while the menu is active.

When the Upgrade menu is closed, the user is also given a small amount of forgiveness. Instead of the character immediately stopping in its place when the menu is closed, the character will slide for a bit until they completely stop or until the user begins inputting a direction. This greatly increases the "coolness" effect when accessing the Upgrade menu.

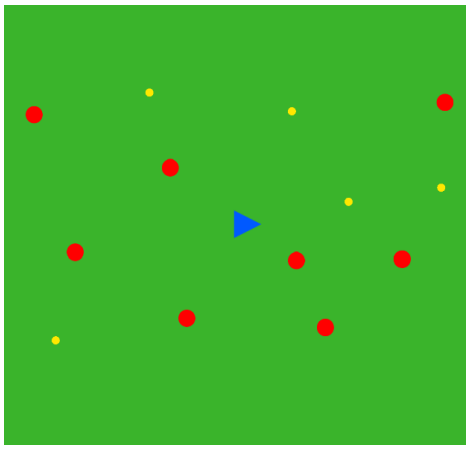


In order to create the playable ground for the game, tilemaps were reused in order to create a large square. To prevent the player from escaping this ground, Unity's Box Colliders were placed along the edges of the square.

To create the minimap, a secondary camera was created. All objects on the scene have a simple sprite image attached to it. This sprite image is set to the 'MinimapOnly' layer. The secondary camera has been specifically set to only render objects on the 'MinimapOnly' layer. By adjusting the camera to output to a game sprite, we are able to create a minimap that will only show simple

shapes which is all the detail required. The following image is a demo of what is observed by the secondary camera.

The player is indicated by a blue triangle as it shows the direction the player is looking in. Red circles represent the enemies. Yellow circles show the coins on the ground. These colours have specifically chosen to contrast well with the background and be easily identifiable at a glance.



To see the pieces of code that handles the Secret Enemy Rush, please see *'Behind a Secret Enemy Rush'*.

Audio + Attributes

Audio was another major consideration throughout development. Specific audio cues make it easier for the user to identify when certain actions have been completed or whether an action requires urgent action. And background music helps to add to the atmosphere of the game and to enrich the user experience.

Throughout the UI, audio is played whenever the user makes or confirms an action. This acts as instant feedback so that the user knows that their input has been received and processed. In particular, UI audio is played whenever the user clicks on a button or saves their settings.

Within games, audio is also played whenever the user performs an action. This also acts as feedback to the user but also adds to the visuals on screen. For example, audio is used when the user hits the golf ball or if the Boss shoots a projectile.

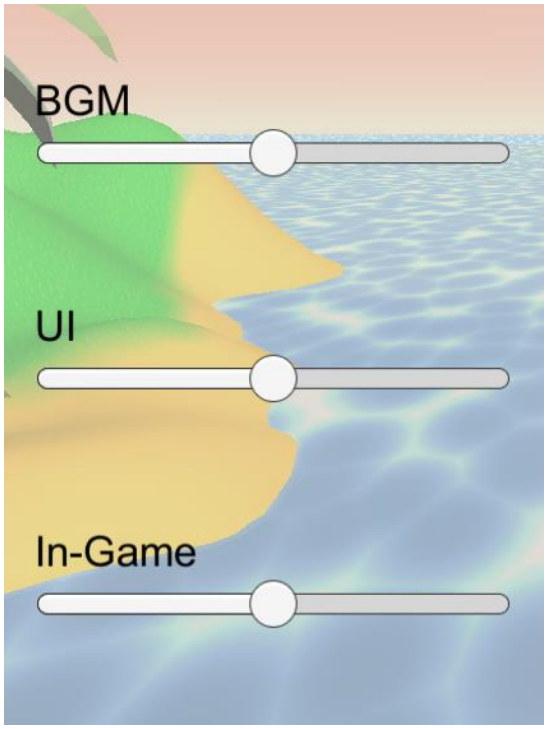
Background music is also available on the main menu and every level in order to add atmosphere to the game and to supplement the theming surrounding each level.

All audio has been specifically chosen as it utilises a Creative Commons or Copyright Free licence and hence is allowed to be used in projects with proper attribution. These attributions can be found in the Settings under “Attributes” but also in the list below.

Main Menu "Nowhere Land" Kevin MacLeod (incompetech.com) Licensed under Creative Commons: By Attribution 4.0 License http://creativecommons.org/licenses/by/4.0/	Credits "Evening Melodrama" Kevin MacLeod (incompetech.com) Licensed under Creative Commons: By Attribution 4.0 License http://creativecommons.org/licenses/by/4.0/	Tutorial "Ambler" Kevin MacLeod (incompetech.com) Licensed under Creative Commons: By Attribution 4.0 License http://creativecommons.org/licenses/by/4.0/	Level 1 "Super Friendly" Kevin MacLeod (incompetech.com) Licensed under Creative Commons: By Attribution 4.0 License http://creativecommons.org/licenses/by/4.0/
Level 2 "Angevin B" Kevin MacLeod (incompetech.com) Licensed under Creative Commons: By Attribution 4.0 License http://creativecommons.org/licenses/by/4.0/	Level 3 "Gloom Horizon" Kevin MacLeod (incompetech.com) Licensed under Creative Commons: By Attribution 4.0 License http://creativecommons.org/licenses/by/4.0/	Level 4 "Constance" Kevin MacLeod (incompetech.com) Licensed under Creative Commons: By Attribution 4.0 License http://creativecommons.org/licenses/by/4.0/	Final Boss "Obliteration" Kevin MacLeod (incompetech.com) Licensed under Creative Commons: By Attribution 4.0 License http://creativecommons.org/licenses/by/4.0/
UI_beep "Menu Beep.wav" by DrMrSir, https://freesound.org/s/529560/	UI_confirm "UI_2-1 NTFO Triangle(Sytrus,arpegio,multiprocessing,rsmpl).wav" by newlocknew, https://freesound.org/s/515829/	IG_scroll "Positive Blip Effect" by Eponn, https://freesound.org/s/531512/	UI_dialogue "Cartoon UI Back/Cancel" by plasterbrain, https://freesound.org/s/423168/
UI_noti "Notification" by IENBA, https://freesound.org/s/545495/	IG_flag "Drop ball in cup-2.wav" by AGFX, https://freesound.org/s/20429	IG_water "Water Click" by Mafon2, https://freesound.org/s/371274/	IG_bullet "Laser shoot" by MusicLegends, https://freesound.org/s/344310/

IG_rocket "Laser Shoot7" by MusicLegends, https://freesound.org/s/344312/	IG_lazer "LazerCannon.ogg" by mango777, https://freesound.org/s/547441/	IG_warning "Warning Sound" by m_cel, https://freesound.org/s/507906/	IG_ballhit "Golf 1.wav" by zolopher, https://freesound.org/s/75203/
IG_heal "Retro game heal sound" by lulyc, https://freesound.org/s/346116/	IG_hurt "Damage sound effect" by Raclure, https://freesound.org/s/458867/		

Audio has also been sorted into three categories, “UI”, “In-game”, and “BGM”(Background Music) so that users are able to adjust the volume of each category separately through the Settings menu, allowing for further customizability.



Loading Screen

Different Scenes are used to split up different “areas” in Unity. In particular, there are 7 scenes that are used in this game.

“**MainMenu**” is the scene that holds the main menu scenes, as well as the 3D background.

Known as “**SampleScene**” in the Unity Editor but formally called “Tutorial” in Level Select is the Tutorial level for the game.

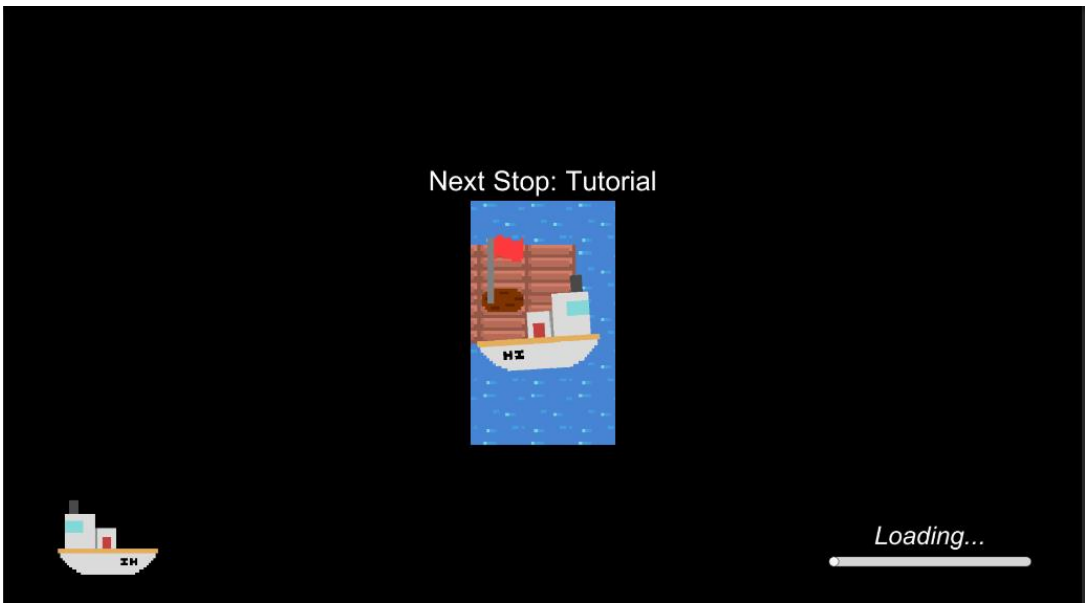
“**LevelOne**”, “**LevelTwo**”, “**LevelThree**”, “**LevelFour**” all are Levels One to Four.

“**TheFinalBoss**” is the scene that contains the Final Boss.

“**FunnyShooter**” contains the Secret Enemy Rush mode.

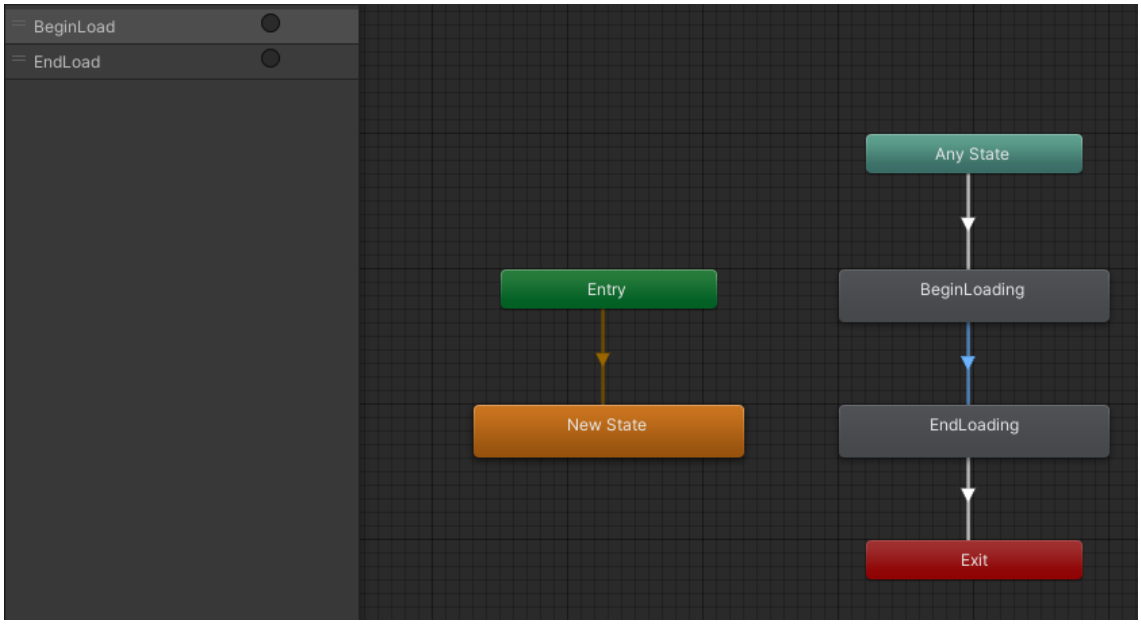
In order to seamlessly transition between each scene, a Scene Transition is used to cover up the switching scenes. This transition is handled by a Unity animation as well as a ‘LoadingScreen’ module that initiates the transition. For details on the code behind this transition, please see “*Transitioning Scenes*”

Depending on the user’s option for “Reduce Motion”, there are two transitions that may play. If the option is disabled, then the transition is a swipe-in from the right before it swipes-out towards the left. If the option is enabled, then the transition will fade-in to black, then fade-out to the new scene. However, instead of just being a blank screen, additional objects were added to this screen.



In order to provide feedback to the user that their game has not frozen, a small animation of a looping boat plays on the bottom left corner. On the bottom right is a loading bar that shows the user the progress of the loaded scene. Finally, the centre of the screen has an image of the scene that the user is going to, as well as the text “Next Stop” which is a play on the fact that boats happen to be an important part of the software.

These features aim to keep the user entertained until the scene is ready to play so that there are no awkward moments of un-loaded assets or poor performance when loading in.



As animations have been used, Unity's "state machines" were also used to trigger the transition in, and transition out of a Scene. The trigger 'BeginLoad' is toggled when the transition needs to be played. The transitioning screen pictured above will stay on screen until another trigger 'EndLoad' is triggered, which is when the Scene has already been loaded. Then the transition to exit will be played and the user will be able to see the proper Scene.

In order to prevent such a jarring effect with the background music, the current audio is progressively lowered to 0 while the scene is loading. Then once the scene is ready, the new background music will progressively get louder until it reaches the user's set audio volume.

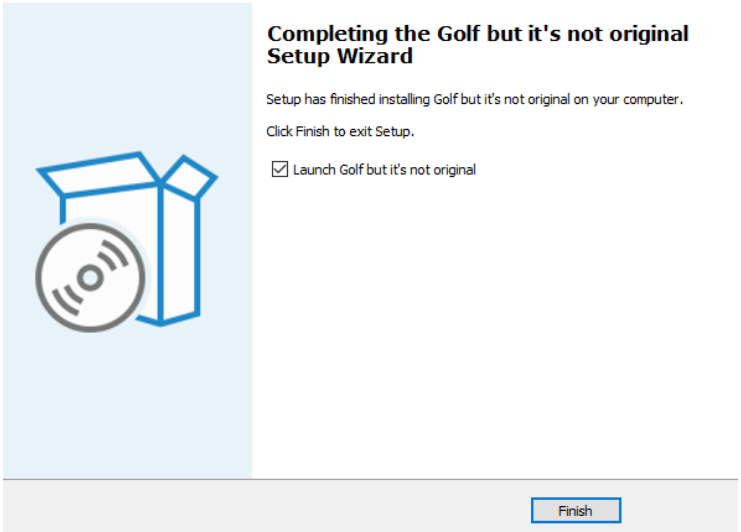
Installation via Inno Setup

While not explicitly mentioned in the requirements, I wanted to package the game inside an installer to offer easy installation, updating, and uninstalling functions to the user. Because Unity games often have multiple required files and folders after they are built, it would be much simpler to offer a single executable that could unpack the game files and create the necessary shortcuts by itself.

Due to time and resource constraints, creating my own installer would not be feasible. After conducting some research, I concluded that the open source program, Inno Setup, would suit my needs. Since this software comes with a wizard, it was extremely easy to package all the Unity executable files into a single installer. And its scripting language was simple to understand to make modifications to the installer process. As a result, the End User Approach is used in this aspect of the solution.

In correspondence with the open source licence of Inno Setup, the software can be found at: https://jrsoftware.org/isinfo.php

After packaging the Unity game files through Inno Setup, a single installer executable was produced which would be distributed to the clients. However, in case the installer failed to install the game properly, the compiled game files would also be offered for manual installation.



Technical Specifications

Throughout development, numerous modules and subroutines have been developed to solve the problem. For ease of understanding the source code, intrinsic documentation has been constantly created alongside the code itself. Intrinsic documentation includes comments to understand the process behind the code, whitespace and indentation for ease of readability, and meaningful identifiers that help other myself and other developers understand the source code.

Saving and Loading

One of the most essential components of the solution is the ability to save and load the user's progress as they play through the game. This would allow for the user's data and records to remain even after they close the game.

Most of the following data items are saved to an external file. With an expanded explanation of each of them below.

```
public List<LevelFormat> LevelData; //Holds data about every level (This data is saved)
public bool BossLevelUnlocked; //Flag for whether the user has unlocked the Boss Level yet (Also saved)

public int TimesPlayedSolo; //Number of times the user has cleared a course by themselves
public int TimesPlayedMulti; //Number of times user has cleared a course in multiplayer

public int BallSkin = 0; //Migrate ball skins from playerPrefs to internal cause cheaters
```

```
//Holds the sprites and status of unlockable balls
public Sprite[] BallSkins;
public bool[] UnlockedBallSkins;
public List<int> LockedBalls;

public bool FullCleared = false; //Has the user completed everything in the game

public float SparkleColour = 0; //The Hue value for the colour of sparkles
```

"**LevelData**" is a List of a custom Class 'LevelFormat' that holds the name of the level, any records that have been set, as well as a replay of the inputs made by the user.

```
//Holds data for the courses
[System.Serializable]
6 references
public class LevelFormat
{
    public string LevelName = "dummy"; //The name saved in Unity
    public string ExternalName = ""; //The name that should show up everywhere else
    public int LevelInt = -99; //The order it's supposed to show in
    public float BestTime = 0; //Time record
    public int BestHits = 0; //Hit record
    public int CollectableGet = 0; //Whether the user has collected the course's collectable. 0 = No, 1 = Just got, 2 = Already get
    public List<GhostData> ghostData; //List of inputs made by the user to be replayed
}
```

"**BossLevelUnlocked**" is a boolean value that acts as a flag. By default, it is false. And it is toggled to become true once the user has completed all golf courses at least once.

"**TimesPlayedSolo**" and "**TimesPlayedMulti**" are integer values that hold how many times the user has either played Singleplayer or Multiplayer. Any time a course is completed, these integers are incremented accordingly.

"**BallSkin**" is another integer value that holds the current skin that the user is using.

"**UnlockedBallSkins**" is a List of boolean values that keep track of whether the user has already unlocked a skin or not. If a certain value is true, that means the corresponding skin has already been unlocked and is available to use.

"**FullCleared**" is another boolean value. Like "BossLevelUnlocked", it acts as a flag and only becomes true once the user has played every level and collected every scroll in the game.

"**SparkleColour**" is a float that stores a value corresponding to the Hue of a HSV colour. This value is only required once the user has Full Cleared the game as it cannot be altered before that point.

In order to save these items, they must first be made "System Serializable". In other words, they must be able to be represented as basic data types so that they can be expressed as a series of bytes. As the majority of the above data is boolean, integer, or float, they are able to be "Serialised".

Once the program receives a call to save the game, it will first be handled by the 'GameManager', which is where all this data is stored during runtime. Specifically, a subroutine called 'SavePlayer()' will be called to save the game.

```
//Save manager stuffs
[ContextMenu("Force Save")]
8 references
public void SavePlayer()
{
    //Package savedata from GM into PlayerData data
    PlayerData data = new PlayerData(this);

    //Send data to SaveSystem
    SaveSystem.SaveGame(data);

    LastSaved = "Game saved at: " + System.DateTime.Now;
    Debug.Log(LastSaved);
}
```

This subroutine will first take all the data from 'GameManager' and then send it to the module 'SaveSystem' and the subroutine 'SaveGame(data)' with the argument 'data' corresponding to the player's data that will be saved.

```
//To save the game. Convert data to binary then save to file
1 reference
public static void SaveGame(PlayerData data)
{
    //Set streams
    BinaryFormatter formatter = new BinaryFormatter();
    MemoryStream ms = new MemoryStream();

    //Serialise, convert to bytes, then checksum
    formatter.Serialize(ms, data);
    byte[] bytes = ms.ToArray();
    var hash = GetMD5Checksum(bytes);
    Debug.Log("Hash for Save File is = " + System.BitConverter.ToString(hash).Replace("-", ""));
    ms.Close();

    //Append checksum to file
    byte[] appended = new byte[hash.Length + bytes.Length];
    System.Buffer.BlockCopy(hash, 0, appended, 0, hash.Length);
    System.Buffer.BlockCopy(bytes, 0, appended, hash.Length, bytes.Length);

    //Check that hash was written correctly
    byte[] readHash = new byte[16];
    System.Buffer.BlockCopy(appended, 0, readHash, 0, readHash.Length);
    Debug.Log("Hash from inside the save file is = " + System.BitConverter.ToString(readHash).Replace("-", ""));

    //Get path ready then save file
    string path = Application.persistentDataPath + "/playerData.golf";
    File.WriteAllBytes(path, appended);
}
```

Firstly, this procedure will open a BinaryFormatter and MemoryStream. A BinaryFormatter is required to convert all the data into binary. And a MemoryStream is used to store the binary data in memory temporarily.

With the BinaryFormatter, the 'data' is Serialised into the MemoryStream and then converted into an array of bytes. This step is essential as it allows us to add further security to the save file in the form of a checksum. A checksum is a series of bytes that have been calculated from some data. And for identical pieces of data, this checksum will always be identical. Therefore, we are able to detect whether a file has been tampered with or is corrupted by whether the checksum of the save data remains consistent.

Hence using the array of bytes, we are able to generate a checksum using the MD5 algorithm with the following function.

```
//Get a cryptographically generated set of bytes that correspond to the data inside the save file
//Ensures data hasn't been majorly been corrupted or tampered with
2 references
public static byte[] GetMD5Checksum(byte[] stream)
{
    using (var md5 = System.Security.Cryptography.MD5.Create())
    {
        var hash = md5.ComputeHash(stream);
        return hash;
        //return System.BitConverter.ToString(hash).Replace("-", "");
    }
}
```

Once this checksum has been generated, we are able to join the set of bytes for the checksum with the bytes representing the save data into another array of bytes called 'appended'. With this 'appended' array of bytes, we are able to directly write it into the following location:

`%userprofile%\AppData\LocalLow\`
`One person, not enough time\Golf but it's not original\playerData.golf`

Once a user's save file has been saved, it can be loaded back into the game by reversing the process. This process is first started with 'GameManager' and 'LoadPlayer()'. This routine then immediately makes another call to 'SaveSystem' to read the file and load the data.

```
[ContextMenu("Force Load")]
2 references
public void LoadPlayer()
{
    //Get savedata from SaveSystem
    PlayerData data = SaveSystem.LoadGame();
```

```
//Load the game. Open file, convert to data then return data
1 reference
public static PlayerData LoadGame()
{
    string path = Application.persistentDataPath + "/playerData.golf";
    if (File.Exists(path))
    {
```

Before trying to access the file, 'LoadGame()' will first check whether the file exists in its expected location. If it does not, it will return an error and prompt the game to create a new save file instead. If the file does exist, then it will continue onto the following steps.

```
//Set binary formatter and FileStream
BinaryFormatter formatter = new BinaryFormatter();

//Open the file as bytes
byte[] bytes = File.ReadAllBytes(path);

//Copy first 16 bytes from File and copy into readHash
byte[] readHash = new byte[16];
System.Buffer.BlockCopy(bytes, 0, readHash, 0, readHash.Length);
var readHashString = System.BitConverter.ToString(readHash).Replace("-", "");
Debug.Log("Hash from inside the save file is = " + System.BitConverter.ToString(readHash).Replace("-", ""));

//Copy rest of File into readRest
byte[] readRest = new byte[bytes.Length - readHash.Length];
System.Buffer.BlockCopy(bytes, readHash.Length, readRest, 0, readRest.Length);

//Get checksum from readRest
var fileHash = GetMD5Checksum(readRest);
var fileHashString = System.BitConverter.ToString(fileHash).Replace("-", "");
Debug.Log("Hash for Loaded File is = " + System.BitConverter.ToString(fileHash).Replace("-", ""));
```

First, it opens a BinaryFormatter stream so that it will be able to Deserialize the series of bytes. Essentially converting the bytes back into meaningful data. Then it will open the file and read all the bytes into an array of bytes called ‘bytes’. As this file contains both the player’s data and checksum, it must be separated before anything else can be done.

As it is already known that an MD5 checksum will always be 16 bytes in length, we are able to precisely extract the first 16 bytes from the raw data, which will be saved into ‘readHash’. Afterwards, we are able to extract the remainder of the data which will be stored in ‘readRest’. Using ‘readRest’, we are able to send it through the checksum function again which will return a set of bytes called ‘fileHash’

```
//Compare checksums
if (fileHashString == readHashString)
{
    //If valid, load game normally
    //Take bytes into stream, then deserialise it as PlayerData
    using (MemoryStream memStream = new MemoryStream(readRest))
    {
        PlayerData data = (PlayerData)formatter.Deserialize(memStream);
        Debug.Log("Data loaded successfully!");
        LoadStatus = 0;
        return data;
    }

} else
{
    //If invalid, don't do anything and return null to force new save file
    Debug.Log("File Checksum invalid. Creating new Save File");
    LoadStatus = 1;
    return null;
}
```

Finally, we are able to compare the two checksums. If they match, we can be certain that the data is valid and it can be returned to ‘GameManager’ to continue processing the data. However, if they do not match, then an error will be returned instead and the game will be prompted to create a new save file.

```
if (data == null)
{
    switch (SaveSystem.LoadStatus)
    {
        case 1: //Hash Error
            ErrorStatus = 1;
            break;

        case 2: //File not exists
            SavePlayer(); //Assume fresh save file
            break;

        default: //Unknown error
            ErrorStatus = 99;
            break;
    }

} else
{
    ErrorStatus = 0;

    //Insert back into GM
    LevelData = data.LevelData;
    BossLevelUnlocked = data.BossLevelUnlocked;
    TimesPlayedSolo = data.TimesPlayedSolo;
    TimesPlayedMulti = data.TimesPlayedMulti;
    UnlockedBallSkins = data.UnlockedBallSkins;
    BallSkin = data.BallSkin;
    FullCleared = data.FullCleared;
    SparkleColour = data.SparkleColour;

    Debug.Log("Game loaded at: " + System.DateTime.Now);
}
```

Back in ‘GameManager’, now that it is certain that the data inside the file was valid, it will be inputted into the variables and the rest of the game can continue loading.

As ‘GameManager’ is a global module and its variables can be accessed from anywhere in the system, it is an extremely versatile tool in holding important data and information related to the current game session and player data.

UI, Behaviour and Buttons

The following pieces of code help manage the behaviour of UI on main and pause menus. As the software supports mouse, keyboard, and controller input, it is essential that the user can seamlessly transition between each input type when navigating main menus.

```
[HideInInspector]
public Vector2 LeftMove;

//Gets value of movement stick
0 references
public void UpDownLeftRight(InputAction.CallbackContext value)
{
    LeftMove = value.ReadValue<Vector2>();
}
```

```
//Any time the user uses arrow keys, make sure FirstSeleced isn't missing, set the current item to the FirstSelected item, and or set the current item
   to FirstSelected. Handles wacky mouse to controller behaviour
Unity Message | 0 references
private void Update()
{
    if (eventSystem.firstSelectedGameObject == null)
    {
        eventSystem.firstSelectedGameObject = eventSystem.currentSelectedGameObject;
    }

    if ((LeftMove != Vector2.zero) && (eventSystem.currentSelectedGameObject != null))
    {
        eventSystem.firstSelectedGameObject = eventSystem.currentSelectedGameObject;
    }

    if ((LeftMove != Vector2.zero) && (eventSystem.currentSelectedGameObject == null || eventSystem.currentSelectedGameObject.activeSelf))
    {
        eventSystem.SetSelectedGameObject(eventSystem.firstSelectedGameObject);
    }
}
```

In order to allow the user to deselect any UI elements using a mouse, then continue navigation using the Arrow Keys on a Keyboard, the last selected object is tracked at all times. This is done by setting ‘firstSelectedGameObject’ to the ‘currentSelectedGameObject’ so that the last selected object is always known. This is done within Unity’s ‘Update()’ function which essentially checks this subroutine once every frame.

Once the user decides to deselect a button using the mouse, then presses a button on the Arrow Keys, another line of code inside the 3rd ‘if’ statement will reassign the ‘currentSelectedGameObject’ to the ‘firstSelectedGameObject’. Essentially resetting the currently selected object to the last object before it was deselected.

These lines of code ensure that navigation throughout the software remains fluid and natural to use no matter the input the user is using.

Another subroutine of code that helps to ensure the fluidity of the menus is ‘BackingOut()’. This subroutine linearly checks every available UI screen before returning to the previous screen. This is the equivalence of pressing ESC or ‘Back’ to return to the previous menu.

```
2 references
public void BackingOut()
{
    if (MainMenu.activeSelf)
    {
        //Dev message panel
        if (DevMsgPanel.activeSelf)
        {
            ReturnDevMsg();
        }
    }
    else if (MultiplayerSelect.activeSelf)
    {
        ReturnMainFromMulti();
    }
    else if (Settings.activeSelf) //If it's the settings
    {
        if (ConfirmRevertScreen.activeSelf) //Check the res menu isn't open. if it is, cancel the res screen instead
        {
            settingsManager.RevertButton();
        }
        else if (DeleteConfirmScreen.activeSelf) //Or else, check it isn't the delete screen. Cancel the delete instead
        {
            settingsManager.CancelDelete();
        }
        else if (CreditsScreen.activeSelf)
        {
```

```
        else if (CreditsScreen.activeSelf)
        {
            settingsManager.ReturnFromCredits();
        }
        else if (AttributeScreen.activeSelf)
        {
            settingsManager.CloseAttributes();
        }
        else //If none those screens are open
        {
            if (settingsManager.WindowMode.gameObject.transform.childCount == 3 && settingsManager.WindowSize.gameObject.transform.childCount == 3 &&
              settingsManager.InputType.gameObject.transform.childCount == 3) //Make sure a dropdown isn't open
            {
                PressReturnToMain(); //Then exit the screen
            }
        }
    }
    else if (RecordsScreen.activeSelf)
    {
        if (ScrollStoryScreen.activeSelf)
        {
            recordManager.ClickCloseScroll();
        }
        else
        {
            ReturnFromRecords();
        }
    }
    else if (LevelSelectScreen.activeSelf && GameManager.GM.SingleMode)
    {
        if (LevelSelectGhostPanel.activeSelf)
```

```
            levelManager.CancelButton();
        }
        else if (LevelSelectHowToBoss.activeSelf)
        {
            levelManager.HowBossOK();
        }
        else
        {
            ReturnFromLevelSelect();
        }
    }
    else if (LevelSelectScreen.activeSelf && !GameManager.GM.SingleMode)
    {
        if (LevelSelectGhostPanel.activeSelf)
        {
            levelManager.CancelButton();
        }
        else if (LevelSelectHowToBoss.activeSelf)
        {
            levelManager.HowBossOK();
        }
        else
        {
            LevelSelectToMultiplayer();
        }
    }
    else if (ReturnUIButton.gameObject.activeSelf)
    {
        PressReturnUI();
    }
```

This final essential UI subroutine handles the screen transitions when moving from one screen to another. Even though there are multiple routines for each direction, they are all virtually identical to one another and so only one will be described here.

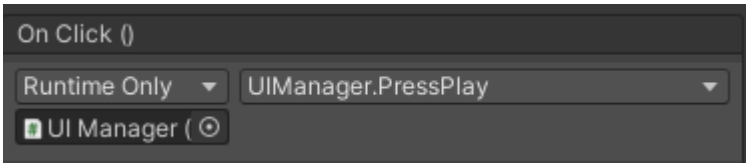
```
2 references
IEnumerator SwipeLeft(RectTransform oldScreen, RectTransform newScreen, Selectable setButton)
{
    eventSystem.SetSelectedGameObject(null);

    float time = 0;
    float Duration = 0.4f;
    float newValue = 923.7604f;
    float otherTargetValue = -923.7604f;
    float oldValue = 0;

    float NewScreenLocation;
    float OldScreenLocation;

    newScreen.gameObject.SetActive(true);

    if (PlayerPrefs.GetInt("ReduceMotion", 0) == 0) //If false
    {
        while (time < Duration)
        {
            OldScreenLocation = Mathf.Lerp(oldValue, otherTargetValue, time / Duration);
            oldScreen.offsetMin = new Vector2(OldScreenLocation, 0);
            oldScreen.offsetMax = new Vector2(OldScreenLocation, 0);

            NewScreenLocation = Mathf.Lerp(newValue, oldValue, time / Duration);
            newScreen.offsetMin = new Vector2(NewScreenLocation, 0);
            newScreen.offsetMax = new Vector2(NewScreenLocation, 0);

            time += Time.deltaTime;
            yield return null;
        }
    }
```

‘SwipeLeft()’ is an IEnumerator. Essentially, it allows for code to be paused and executed over multiple frames. In contrast to a regular subroutine where everything is processed in a single frame. With all the arguments passed in from the calling subroutine, first it sets all the variables that are required. Then it checks if the accessibility option “Reduce Motion” is enabled or not. If it is, then it will skip over the animation that plays. If it is not, then it will linearly slide the screen across as can be seen in the final build of the game.

```
    oldScreen.offsetMin = new Vector2(otherTargetValue, 0);
    oldScreen.offsetMax = new Vector2(otherTargetValue, 0);

    newScreen.offsetMin = new Vector2(0, 0);
    newScreen.offsetMax = new Vector2(0, 0);

    oldScreen.gameObject.SetActive(false);

    setButton.Select();
    yield return null;
    if (inputSystem.currentControlScheme == "Controller")
    {
        setButton.Select();
        eventSystem.firstSelectedGameObject = eventSystem.currentSelectedGameObject;
    }
    else
    {
        eventSystem.firstSelectedGameObject = setButton.gameObject;
        eventSystem.SetSelectedGameObject(null);
    }
}
```

After the screens are in their final locations, the first UI element for that screen will be selected or prepared so that the user can easily continue navigating the UI.

In order to allow the user to navigate between different screens, Unity buttons are utilised so that a press of a button will activate some action. Inside Unity, subroutines can be attached to buttons and other UI elements so that the developer can specify the action that is performed when a selection is made.



In the main menu, multiple small subroutines have been made to facilitate movement between different screens, as well as setting up variables and other modules.

```
//Level Select button
0 references
public void PressPlay()
{
    GameManager.GM.SingleMode = true;
    GameManager.GM.NumPlayers.Add(new MultiPlayerClass { PlayerIndex = 0, AimingSensitivity = PlayerPrefs.GetFloat("Sensitivity", 4) });
    MultiSelectScript.CurrentlyLoading = true;

    AudioManager.instance.PlaySound("UI_beep");
    StartCoroutine(SwipeUp(MainMenuRect, LevelSelectRect, LevelSelectFirstButton));
    levelManager.enabled = true;
}

0 references
public void PressPlayGhost()
{
    GameManager.GM.SingleMode = true;
    GameManager.GM.GhostMode = true;
    GameManager.GM.NumPlayers.Add(new MultiPlayerClass { PlayerIndex = 0, AimingSensitivity = PlayerPrefs.GetFloat("Sensitivity", 4) });
    MultiSelectScript.CurrentlyLoading = true;

    AudioManager.instance.PlaySound("UI_beep");
    StartCoroutine(SwipeUp(MainMenuRect, LevelSelectRect, LevelSelectFirstButton));
    levelManager.enabled = true;
}
```

Both ‘PressPlay()’ and ‘PressPlayGhost()’ enable the user to move to the Level Select screen. Although they affect different variables, they will still play some audio as well as moving the screens.

```
//Multiplayer button
0 references
public void PressPlay2P()
{
    inputSystem.enabled = false;

    inputManager.enabled = true;
    inputManager.EnableJoining();

    AudioManager.instance.PlaySound("UI_beep");
    StartCoroutine(SwipeDown(MainMenuRect, MultiplayerRect, MultiplayerFirstButton));
}
```

Meanwhile, ‘PressPlay2P()’ is for the Multiplayer button and will enable other modules and systems to prepare for additional players to the game.

```
//Settings button
0 references
public void PressSettings()
{
    AudioManager.instance.PlaySound("UI_beep");
    StartCoroutine(SwipeLeft(MainMenuRect, SettingsRect, SettingsFirstButton));
}

//Records button
0 references
public void PressRecords()
{
    recordManager.UpdateThings();
    AudioManager.instance.PlaySound("UI_beep");
    StartCoroutine(SwipeRight(MainMenuRect, RecordsRect, RecordsFirstButton));
}
```

Other buttons such as the one going to Settings and Records function a bit more primitively, however they still perform essential tasks for the navigation of UI.

```
//Quit button
0 references
public void PressQuit()
{
    #if UNITY_EDITOR
            EditorApplication.isPlaying = false;
    #else
            Application.Quit();
    #endif
}
```

The Quit button is a special case. As for debugging purposes, special expressions are used to close the application in different ways whether or not it is running inside the Unity Editor. If it's running inside the Unity Editor, then the editor is asked to stop running the application. If not, then the application will just close as normal.

In addition to these buttons and many more scattered across the software, there are also buttons that return to a previous menu. In particular, due to Level Select being accessible from Singleplayer or Multiplayer, a special case was made so that it could return to the intended menu.

```
//Returning from Level Select
1 reference
public void ReturnFromLevelSelect()
{
    if (GameManager.GM.SingleMode)
    {
        GameManager.GM.NumPlayers.Clear();

        MultiSelectScript.ResetActions();

        if (!GameManager.GM.GhostMode)
        {
            StartCoroutine(SwipeDown(LevelSelectRect, MainMenuRect, PlayButton)); //Not ghost mode
        } else
        {
            StartCoroutine(SwipeDown(LevelSelectRect, MainMenuRect, PlayGhostButton)); //Ghost mode
        }

        GameManager.GM.SingleMode = false;
        GameManager.GM.GhostMode = false;

        MultiSelectScript.CurrentlyLoading = false;

        AudioManager.instance.PlaySound("UI_beep");

        levelManager.enabled = false;
    } else
    {
        LevelSelectToMultiplayer();
    }
}
```

Using ‘GameManager’ as a way to store the current mode that the user is accessing, we are able to check whether the user is returning from Singleplayer, Vs. Ghost, or Multiplayer mode. Using some ‘if’ statements to check the current mode, we are able to reset variables, as well as redirect processing to another subroutine.

```
3 references
public void LevelSelectToMultiplayer()
{
    MultiSelectScript.CurrentlyLoading = false;
    inputManager.EnableJoining();
    AudioManager.instance.PlaySound("UI_beep");
    StartCoroutine(SwipeDown(LevelSelectRect, MultiplayerRect, MultiplayerFirstButton));
    levelManager.enabled = false;
}
```

As seen above, if the current mode is Singleplayer or Vs. Ghost, then Level Select will return to the Main Menu. However, if the current mode is Multiplayer, then a different subroutine is used to return to the Multiplayer menu instead.

In order to go from a course straight to level select, a special check was also made for that scenario.

```
//Subscribes or unsubscribes from sceneLoaded, handles CourseSelect
Unity Message | 0 references
private void OnEnable()
{
    SceneManager.sceneLoaded += CheckLevelSelect;
}
Unity Message | 0 references
private void OnDisable()
{
    SceneManager.sceneLoaded -= CheckLevelSelect;
}
```

When the UI first loads, it will ‘subscribe’ to a certain event and if that event runs, then it will call a specific subroutine. In this case, if the Scene is Loaded, then it will run ‘CheckLevelSelect’.

```
//If returning from a Course to Course Select screen, this will load
2 references
public void CheckLevelSelect(Scene scene, LoadSceneMode mode)
{
    if (GameManager.GM.LoadIntoLevelSelect)
    {
        GameManager.GM.LoadIntoLevelSelect = false;

        MainMenu.SetActive(false);
        LevelSelectScreen.SetActive(true);

        MainMenuRect.offsetMin = new Vector2(-519.62f, 0);
        MainMenuRect.offsetMax = new Vector2(-519.62f, 0);

        LevelSelectRect.offsetMin = new Vector2(0, 0);
        LevelSelectRect.offsetMax = new Vector2(0, 0);

        levelManager.enabled = true;

        if (GameManager.GM.SingleMode == false)
        {
            MultiSelectScript.StraightIntoLevelSelect();
        }

        LevelSelectFirstButton.Select();
        eventSystem.firstSelectedGameObject = eventSystem.currentSelectedGameObject;
    }
}
```

Due to limitations in Unity, you are unable to pass data to different “Scenes” directly, like parameters and subroutines. Instead, scripts/modules are able to persist through different Scenes and thus, data can be stored in scripts to be passed onto the new Scene. As the Main Menu and courses exist in different Scenes, a flag from the course is set in ‘GameManager’. This flag indicates that the user intends to return to Level Select instead of the Main Menu.

Once the UI loads, it checks for this flag. If it is set to true, then the required variables will be properly set and the user will land at Level Select. If this flag isn’t set, then it is assumed that the user wants to return to the Main Menu and thus, everything continues as if the variables have been reset.

Managing Settings

Although the majority of player data is considered critical and therefore is saved in an external file, player's preferences are not as critical and hence, can be stored in a less secure location called 'PlayerPrefs'. To be precise, 'PlayerPrefs' refers to a location within the Windows Registry. These are the following settings that are saved in 'PlayerPrefs'.

```
OldWindow = PlayerPrefs.GetInt("WindowMode", 0);
OldResolution = PlayerPrefs.GetInt("WindowSize", 0);
OldInputType = PlayerPrefs.GetInt("InputType", 0);
OldSensitivity = PlayerPrefs.GetFloat("Sensitivity", 4);
OldDebugWindow = IntToBool(PlayerPrefs.GetInt("DebugWindow", 0));
OldBGM = PlayerPrefs.GetFloat("BGM", 5f);
OldUI = PlayerPrefs.GetFloat("UI", 5f);
OldInGame = PlayerPrefs.GetFloat("InGame", 5f);
OldColourPicker = GameManager.GM.SparkleColour;
OldReduceMotion = IntToBool(PlayerPrefs.GetInt("ReduceMotion", 0));
OldCRT = IntToBool(PlayerPrefs.GetInt("CRT", 0));
OldShowFrames = IntToBool(PlayerPrefs.GetInt("ShowFrames", 0));
OldFPS = PlayerPrefs.GetInt("FPS", 1);
OldvSync = IntToBool(PlayerPrefs.GetInt("vSync", 1));
```

"**WindowMode**" stores an integer representation of the current Window Mode that the user is using. This is either Windowed Fullscreen or Windowed.

"**WindowSize**" also stores an integer representation of the current Resolution that the user is using. In order of largest to smallest, the resolutions that are supported by the software are: 1920x1080, 1600x900, 1280x720, 1024x576, 960x450 and 800x600.

"**InputType**" stores an integer representation of the current Input Type that the user is using. This is either 'Mouse and Drag' or 'Keyboard and Buttons'.

"**Sensitivity**" is a float that stores the player's sensitivity setting when they are using Button controls. It directly correlates to the speed at which the aiming reticle will rotate when aiming the ball.

"**DebugWindow**" is supposed to be stored as a boolean. However as 'PlayerPrefs' does not support boolean values, a function called 'IntToBool' and 'BoolToInt' converts the boolean flag into an integer representation.

"**BGM**", "**UI**", and "**InGame**" are all floats that store the audio volume levels for their respective channels.

"**ReduceMotion**" is another boolean value that is represented as an integer using functions. This indicates whether fast moving objects should be slowed down or not for greater accessibility options.

"**CRT**" is another boolean value that is represented as an integer. This indicates whether the 'Retro Mode' setting should be enabled or not.

"**ShowFrames**" is another boolean value that indicates whether the frames per second counter should be enabled.

"**FPS**" is an integer value that holds the current Frame Rate that the user has set.

"**vSync**" is a boolean value that determines whether vSync should be enabled. vSync is rendering technology that prevents issues such as screen tearing from occurring.

By using'PlayerPrefs.Get_()', 'PlayerPrefs.Set_()' and 'PlayerPrefs.Save()', we are able to read and write these values and store user preferences related to the program. Modules from anywhere in the system are also available to access these values, which makes it an extremely valuable aspect of the solution.

In order to allow for forgiveness when the user changes a critical graphics option, a countdown is initiated where the user has 10 seconds to confirm before the setting is reverted. This countdown occurs when the user is backing out from the Settings and ensures any screen issues can be reverted before they are saved.

```
//Countdown timer before reverting screen options
float currentCountdownValue;
1 reference
public IEnumerator DisplayCountdown(float countdownValue)
{
    eventSystem.SetSelectedGameObject(TheConfirmButton.gameObject);
    eventSystem.firstSelectedGameObject = TheConfirmButton.gameObject;

    currentCountdownValue = countdownValue;

    ChangedOptionsText.text = "";
    if (OldWindow != WindowMode.value)
    {
        ChangedOptionsText.text += "Window Mode, ";
    }
    if (OldResolution != WindowSize.value)
    {
        ChangedOptionsText.text += " Window Size, ";
    }
    if (OldFPS != FPSSlider.value)
    {
        ChangedOptionsText.text += "Frame Rate, ";
    }
    if (OldvSync != vSyncToggle.isOn)
    {
        ChangedOptionsText.text += "vSync, ";
    }
    ChangedOptionsText.text = ChangedOptionsText.text.Remove(ChangedOptionsText.text.Length - 2, 2);

    while (currentCountdownValue >= 0)
    {
        CountdownText.text = "Graphics will revert in " + currentCountdownValue.ToString() + " seconds";
        yield return new WaitForSeconds(1f);
        currentCountdownValue--;

        if (currentCountdownValue == 0)
        {
            ForcedOverride = true;

            WindowMode.value = OldWindow;
            WindowSize.value = OldResolution;
            FPSSlider.value = OldFPS;
            vSyncToggle.isOn = OldvSync;

            ForcedOverride = false;

            RevertWindowed();

            ConfirmRevertScreen.SetActive(false);
            uiManager.PressReturnToMain();

            StopCoroutine(ScreenCheck);
        }
    }
}
```

To implement this countdown timer, another IEnumerator is used as it would be able to count the time elapsed over multiple frames. The line ‘yield return new WaitForSeconds(1f)’ pauses execution of the module for 1 second before it continues in its ‘while’ loop. This allows us to implement a countdown all the way to 0 and by that point, the setting will automatically be reverted.

```
//Confirming display options buttons
0 references
public void ConfirmButton()
{
    StopCoroutine(ScreenCheck);
    ConfirmRevertScreen.SetActive(false);

    OldWindow = WindowMode.value;
    OldResolution = WindowSize.value;
    OldFPS = (int)FPSSlider.value;
    OldvSync = vSyncToggle.isOn;

    PlayerPrefs.SetInt("WindowMode", WindowMode.value);
    PlayerPrefs.SetInt("WindowSize", WindowSize.value);
    PlayerPrefs.SetInt("FPS", (int)FPSSlider.value);
    PlayerPrefs.SetInt("vSync", BoolToInt(vSyncToggle.isOn));
    PlayerPrefs.Save();

    GameManager.GM.gameObject.GetComponent<DebugLogCallbacks>().UpdatePlayPrefsText();

    uiManager.PressReturnToMain();
}
```

If the user presses the ‘Confirm’ button, then the IEnumerator is stopped and the option is saved as normal.

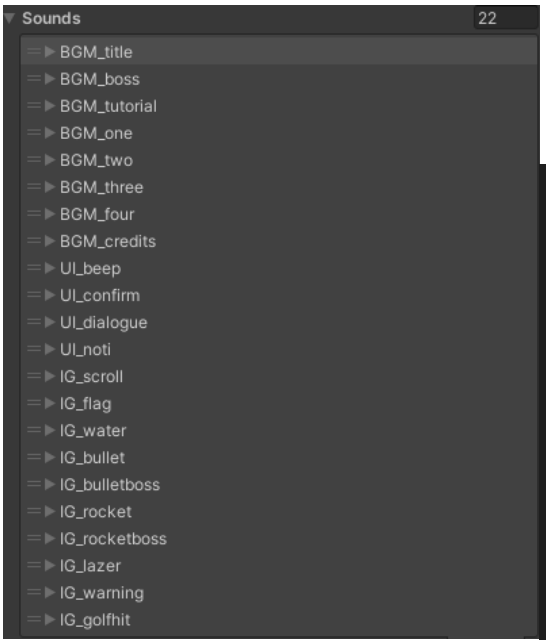
```
1 reference
public void RevertButton()
{
    StopCoroutine(ScreenCheck);
    RevertWindowed();
    ConfirmRevertScreen.SetActive(false);

    ForcedOverride = true;

    WindowMode.value = OldWindow;
    WindowSize.value = OldResolution;
    FPSSlider.value = OldFPS;
    vSyncToggle.isOn = OldvSync;

    ForcedOverride = false;

    uiManager.PressReturnToMain();
}
```

If time runs out or Revert is pressed, the settings will be reset instead.

Playing Audio

As audio is a constant throughout software, specific modules were designed so that audio could be played, paused, or adjusted from anywhere in the system.

Inside Unity, every unique audio file has been added to an ‘AudioManager’ that holds information about each file.




```
[System.Serializable]
8 references
public class Sound
{
    public string name;
    public AudioClip clip;
    [Range(0, 2)]
    public int audioPurpose; //0 = bgm, 1 = ui, 2 = in-game

    [Range(0f, 1f)]
    public float volume = 1;
    [Range(0.1f, 3f)]
    public float pitch = 1;
    public bool loop;

    [HideInInspector]
    public AudioSource source;
}
```

A custom Class called ‘Sound’ holds information including the name of the track, the audio file itself, its purpose (whether BGM, UI, or In-Game), its volume, pitch, whether it’s supposed to loop, and the Unity component that actually outputs the audio.

```
Unity Message | 0 references
private void Awake()
{
    if (instance != null && instance != this)
    {
        Destroy(this.gameObject);
    }
    else
    {
        instance = this;
    }
    DontDestroyOnLoad(this.gameObject);

    foreach (Sound s in sounds)
    {
        s.source = gameObject.AddComponent<AudioSource>();

        s.source.clip = s.clip;
        s.source.volume = s.volume;
        s.source.pitch = s.pitch;
        s.source.loop = s.loop;
    }
}
```

‘Awake()’ is another Unity method that gets called when the script is being loaded. For every audio file in ‘SoundManager’, it will create a new AudioSource with the required properties set up. Once all the AudioSources are ready, any module in the system will be able to play or adjust clips.

```
78 references
public void PlaySound(string name)
{
    Sound s = Array.Find(sounds, sound => sound.name == name);
    if (s == null)
    {
        Debug.Log("Uhoh, can't find " + name + " to play");
        return;
    }

    switch (s.audioPurpose)
    {
        case 0:
            s.source.volume = PlayerPrefs.GetFloat("BGM", 5f) / 10;
            CurrentlyPlayingBGM = s.name;
            break;

        case 1:
            s.source.volume = PlayerPrefs.GetFloat("UI", 5f) / 10;
            break;

        case 2:
            s.source.volume = PlayerPrefs.GetFloat("InGame", 5f) / 10;
            break;

        default:
            break;
    }

    s.source.Play();
}
```

'PlaySound(name)' is used to play any audio that has been loaded into the AudioManager. By using the 'Find' method as well as a Predicate to find the name of a matching sound clip, we are able to adjust the volume of the clip according to its category, and then begin playing it until it ends or until it continues looping.

Using a similar method to 'PlaySound(name)', 'StopPlaying(name)' pauses any audio by finding its clip, and then stopping the source.

```
2 references
public void StopPlaying(string name)
{
    Sound s = Array.Find(sounds, sound => sound.name == name);
    if (s == null)
    {
        Debug.Log("Uhoh, can't find " + name + " to play");
        return;
    }

    s.source.Stop();
}
```

As seen with the blocks of code handling UI buttons, any audio can be easily played by calling the subroutine and passing the name of the audio that needs to be played.

```
AudioManager.instance.PlaySound("UI_beep");
```

While this is an effective method for most audio to be played, it does not allow for multiple instances of the same audio to play at the same time. Therefore, especially for multiplayer sessions where there are multiple players playing at the same time, audio is handled directly by the player's character instead.

Handling Multiple Players

As local Multiplayer support is considered to be a part of the requirements for the solution, it was essential that there were modules to facilitate the adding of new players, assignment of controllers, and finally the loading of all players. As a part of the RAD approach and to apply already existing modules as a part of the solution, Unity's own Player Input System was used as it was an effective, time effective, and cost effective way of allowing multiple players to use the program at once.

```
//Multiplayer button

public void PressPlay2P()
{
    inputSystem.enabled = false;

    inputManager.enabled = true;
    inputManager.EnableJoining();

    AudioManager.instance.PlaySound("UI_beep");
    StartCoroutine(SwipeDown(MainMenuRect, MultiplayerRect, MultiplayerFirstButton));
}
```

Handling multiple players begins at the Multiplayer menu. Once the button to enter the menu is pressed, the subroutine adjusts the systems that are being used. Such as disabling the Input System used for a single player and enabling the Player Input Manager which will allow up to 4 players to join the game.

```
//When the player joins, add them to GameManager, change text, etc etc
0 references
public void WhenAPlayerJoins(PlayerInput value)
{
    string ControlName;
    int ControlType;
    int pIndex = value.playerIndex;

    //Checks if enough indexes are available, if not create new index
    if (GameManager.GM.NumPlayers.Count - 1 < pIndex)
    {
        GameManager.GM.NumPlayers.Add(new MultiPlayerClass { });
    }

    //Checks device type being connected. Applies appropriate text or control type
    if (value.currentControlScheme == "Controller")
    {
        ControlName = "Controller/Buttons";
        ControlType = 2; //2 = Controller buttons

    } else //Or else it's a keyboard user and let them make a choice
    {
        ControlName = "";
        DropdownList[pIndex].gameObject.SetActive(true);
        DropdownList[pIndex].value = 0;
        ControlType = 0;
    }

    //Displays and stores all relevant information
    TextList[pIndex].text = "Connected\n" + ControlName;
    GameManager.GM.NumPlayers[pIndex].PlayerIndex = pIndex;
    GameManager.GM.NumPlayers[pIndex].ControlType = ControlType;
    GameManager.GM.NumPlayers[pIndex].inputDevice = value.GetDevice<InputDevice>();
    GameManager.GM.NumPlayers[pIndex].deviceName = value.GetDevice<InputDevice>().name;
```

At the Multiplayer menu, the Player Input Manager awaits for a button input from any connected peripheral, whether 'Mouse and Keyboard' or 'Controller'. When an input is detected, it will make a call to 'WhenAPlayerJoins(value)'.

This subroutine creates a new entry in GameManager's 'NumPlayers', which is a List that is used to track the currently connected players. Once a new data element has been created in this list, it will check whether the player is using a Controller or not. Within the respective 'if' branches, it will adjust text and objects as required. Once the 'if' statement is exited, 'NumPlayers' is updated with the relevant information including their Player Index, Control Type, Input Device, and Device Name.

```
//Display player's card
PlayerPanels[pIndex].SetActive(true);

//Once more than 1 player is present, allow game to start
if (inputManager.playerCount > 1)
{
    PlayBtn.interactable = true;
    eventSystem.firstSelectedGameObject = dummyObject;
    eventSystem.SetSelectedGameObject(dummyObject);
    PlayText.text = "Ready to Play!";

    print("ping actual play btn");
}
```

Afterwards, the Player's card in the Multiplayer menu is made visible and it is checked whether enough players have joined the game. If there are not enough players yet, it will prevent the users from moving to Level Select.

In the case a player wishes to leave the session, they are able to do so in the Multiplayer menu.

```
//When a controller disconnects or other
0 references
public void WhenAPlayerDisconnects(PlayerInput value)
{
    if (CurrentlyLoading == false) //Needs check in-case game is changing scenes
    {
        //Disable and reset everything
        PlayerPanels[value.playerIndex].SetActive(false);
        DropdownList[value.playerIndex].gameObject.SetActive(false);
        TextList[value.playerIndex].text = "Not Connected";
        GameManager.GM.NumPlayers[value.playerIndex].PlayerIndex = 99;

        if (inputManager.playerCount == 1 || value.playerIndex == 0) //If player left is only 1, disable play button
        {
            PlayBtn.interactable = false;
            //eventSystem.firstSelectedGameObject = null;
            eventSystem.SetSelectedGameObject(null);
            PlayText.text = "Waiting for players...";

            print("ping disabled play btn");
        }
    }
}
```

When the game receives a call to remove a player, and the game isn't switching to another scene, it will first disable the Player's card. Then it will update the relevant entry in 'NumPlayers' with a sentinel value so that it does not mess with the order of currently connected players. Once the player has been removed, the game checks whether there are still enough players to continue. If there are not, then it will disable the Play button until another player joins.

Logic and Input for Golf

When the user starts a game of golf, a number of modules are activated to set up the level before the user starts playing. ‘SpawnBall’ is the module that instantiates the player ball’s before they are given control. The ‘Awake()’ function will instantly run when the Scene is first loaded.

```
Unity Message | 0 references
void Awake()
{
    GameManager gameMan = GameManager.GM;
    SpawnLocation = gameObject;

    if (gameMan.SingleMode == true || gameMan.NumPlayers.Count == 1) //If game is entering with only 1 player, grab their preference
    {
        var InputType = PlayerPrefs.GetInt("InputType", 0);
        var SkinType = gameMan.BallSkin;
        GameObject ABall;

        switch (InputType)
        {
            case 0: //Mouse and drag
                ABall = Instantiate(BallDragPrefab, SpawnLocation.transform.position, Quaternion.identity);
                var BallControllerA = ABall.GetComponent<DragAndAimControllerManager>();
                BallControllerA.BallSprite.sprite = gameMan.BallSkins[SkinType];
                break;

            case 1: //Button and controllers
                ABall = Instantiate(BallButtonPrefab, SpawnLocation.transform.position, Quaternion.identity);
                var BallControllerB = ABall.GetComponent<AltAimControllerManager>();
                BallControllerB.BallSprite.sprite = gameMan.BallSkins[SkinType];
                break;

            default:
                break;
        }

        if (GameManager.GM.GhostMode)
        {
            Instantiate(GhostBallPrefab, SpawnLocation.transform.position, Quaternion.identity);
        }
```

First it will check whether the user is playing in Singleplayer mode. If they are, then it is known that it will only have to spawn in a single player with that player’s preferred skin. Depending on their Input Type, it will spawn in a ball that has the required scripts for that Input Type. Finally, it’ll check whether the game is in Vs. Ghost mode and if it is, it’ll spawn in the Ghost ball.

If the game is in Multiplayer mode instead, a ‘foreach’ loop is used to instantiate a ball for every connected player. Therefore, it would skip over any players that have an index of ‘99’ as they have been marked to have disconnected from the game. This iteration will also swap out the ball’s sprite to one that corresponds to their player index. These sprites can be seen under “*In a Game of Golf*”.

```
switch (item.ControlType) //Depending on their control type, spawn in different ball. And initialise properties
{
    case 0: //Mouse and drag
        CurrentBall = PlayerInput.Instantiate(BallDragPrefab, item.PlayerIndex, null, -1, item.inputDevice);

        var ControllerManagerA = CurrentBall.GetComponent<DragAndAimControllerManager>();

        ControllerManagerA.BallSprite.sprite = MultiSprites[item.PlayerIndex];
        ControllerManagerA.gameObject.transform.position = SpawnLocation.transform.position;
        ControllerManagerA.spriteMask.frontSortingOrder = DefaultSpriteMasks[item.PlayerIndex] + 1;
        ControllerManagerA.spriteMask.backSortingOrder = DefaultSpriteMasks[item.PlayerIndex] - 1;
        ControllerManagerA.insideSprite.sortingOrder = DefaultSpriteMasks[item.PlayerIndex];
        ControllerManagerA.outlineSprite.sortingOrder = DefaultSpriteMasks[item.PlayerIndex];
        ControllerManagerA.insideSprite.color = MultiColours[item.PlayerIndex];
        break;

    case 1: //Keyboard
        CurrentBall = PlayerInput.Instantiate(BallButtonPrefab, item.PlayerIndex, "Keyboard", -1, item.inputDevice);
        CurrentBall.neverAutoSwitchControlSchemes = true;

        var ControllerManagerB = CurrentBall.GetComponent<AltAimControllerManager>();

        ControllerManagerB.BallSprite.sprite = MultiSprites[item.PlayerIndex];
        ControllerManagerB.gameObject.transform.position = SpawnLocation.transform.position;
        ControllerManagerB.spriteMask.frontSortingOrder = DefaultSpriteMasks[item.PlayerIndex] + 1;
        ControllerManagerB.spriteMask.backSortingOrder = DefaultSpriteMasks[item.PlayerIndex] - 1;
        ControllerManagerB.insideSprite.sortingOrder = DefaultSpriteMasks[item.PlayerIndex];
        ControllerManagerB.outlineSprite.sortingOrder = DefaultSpriteMasks[item.PlayerIndex];
        ControllerManagerB.insideSprite.color = MultiColours[item.PlayerIndex];
        break;

    case 2: //Controller
        CurrentBall = PlayerInput.Instantiate(BallButtonPrefab, item.PlayerIndex, "Controller", -1, item.inputDevice);
        CurrentBall.neverAutoSwitchControlSchemes = true;

        var ControllerManagerC = CurrentBall.GetComponent<AltAimControllerManager>();

        ControllerManagerC.BallSprite.sprite = MultiSprites[item.PlayerIndex];
        ControllerManagerC.gameObject.transform.position = SpawnLocation.transform.position;
        ControllerManagerC.spriteMask.frontSortingOrder = DefaultSpriteMasks[item.PlayerIndex] + 1;
        ControllerManagerC.spriteMask.backSortingOrder = DefaultSpriteMasks[item.PlayerIndex] - 1;
        ControllerManagerC.insideSprite.sortingOrder = DefaultSpriteMasks[item.PlayerIndex];
        ControllerManagerC.outlineSprite.sortingOrder = DefaultSpriteMasks[item.PlayerIndex];
        ControllerManagerC.insideSprite.color = MultiColours[item.PlayerIndex];
        break;

    default:
```

At the same time ‘SpawnBall’ is instantiating the game, another module is preparing the timer and internal status about each player. This module is ‘GameStatus’ and holds information about the status of the game, whether the scroll has been collected, as well as ending the game once everyone has reached the goal.

In preparation for the game to begin, a call is made to GameStatus and its ‘Start()’ method. ‘Start()’ is also run when a script is loaded. However, it is performed after ‘Awake()’ but before the first ‘Update()’ function. Therefore in Unity, calls are made in the order of Awake, Start, then Update.

```
//Get required data from GM
Unity Message | 0 references
private void Start()
{
    foreach (var item in GameManager.GM.NumPlayers)
    {
        if (item.PlayerIndex != 99)
        {
            gameStat.playerStatuses[item.PlayerIndex].playerIndex = item.PlayerIndex;
        }
    }
    SingleMode = GameManager.GM.SingleMode;

    if (!SingleMode)
    {
        UIIcon.gameObject.SetActive(false);
        IconPanel.SetActive(false);
    }

    //Get all player inputs and see whether they're player one
    inputs = FindObjectsOfType<PlayerInput>();
    foreach (var item in inputs)
    {
        if (item.playerIndex == 0)
        {
            InputOne = item;
            POneUISys = item.gameObject.GetComponent<MultiplayerEventSystem>();
        }
    }
}
```

First, every player from NumPlayers is synced into a list in GameStatus called “playerStatuses” so that extra information about the player in the stage can be kept. Then it is checked whether the game is in Multiplayer mode, and if it is, it will disable the scroll on the level. Finally, it finds Player One out of all the players and it stores its Input System for future use.

The game does not actually start until a call is made from the Loading Screen manager. This way, the scene transition will have a chance to complete before starting the game. Once the transition has concluded, a call will be made to ‘BeginGame()’

```
//After scene loads, get things loaded
2 references
public void BeginGame()
{
    inputs = FindObjectsOfType<PlayerInput>();

    //If we aren't in tutorial mode, allow ball to move
    if (!GameManager.GM.TutorialMode)
    {
        foreach (var item in inputs)
        {
            item.SwitchCurrentActionMap("In-Game Ball");
        }
    }

    GhostBallMove ghostBallMove = FindObjectOfType<GhostBallMove>();
    //If ghosts are enabled, start the ghost
    if (GameManager.GM.GhostMode && GameManager.GM.SingleMode && ghostBallMove != null)
    {
        ghostBallMove.StartReplay();
    }

    gameStat.StartTimer(); //And start the timer
}
```

The game will keep track of all the Player Inputs that are currently in the game. If the game isn’t in the Tutorial level, it will allow all players to begin moving. If Vs. Ghost Mode is also active, it will instruct the Ghost ball to begin its replay. And finally, it will begin the timer.

```
1 reference
public void StartTimer()
{
    StartCoroutine(IncrementTimer());
}

//Timer will continue until the game is over (exits the loop) or a pause is called (stops counter from incrementing)
1 reference
IEnumerator IncrementTimer()
{
    while (gameStat.GameOver == false)
    {
        if (ForcePause == false)
        {
            gameStat.Timer += Time.deltaTime;
            timertext.text = "Timer: " + gameStat.Timer.ToString("F2");
        }
        yield return null;
    }
}
```

'StartTimer()' is just a subroutine to start another IEnumerator 'IncrementTimer()' that will begin incrementing the timer every frame unless the game is either paused or ended.

If the game is in Vs. Ghost mode, whenever the user does an action, the exact timing, aiming angle, and direction is stored into GameStatus until the end of the game.

```
//Receive hit info from ball and add it to the list
3 references
public void AddGhostData(float Power, float Angle, bool Restart)
{
    IntervalTime = Timer - AccumalativeTime;
    AccumalativeTime = Timer;
    RecordingGhostData.Add(new GhostData{HitAngle = Angle, HitPower = Power, Timing = IntervalTime, ResetPos = Restart});
}
```

Apart from these functions, GameStatus remains dormant until a player reaches the flag. A trigger is established at the flag so once a ball enters a certain area, a call will be made to the ball which will send another call to 'SubmitRecord()'

```
//When flag is hit, pass parametres into list
1 reference
public void SubmitRecord(int pIndex, int NumHits, MoveBall ball)
{
    gameStat.playerStatuses[pIndex].Completed = true;
    gameStat.playerStatuses[pIndex].NumHits = NumHits;
    gameStat.playerStatuses[pIndex].Time = gameStat.Timer;

    ball.UpdateTimerText(gameStat.playerStatuses[pIndex].Time);

    if (NumHits > MaxHits)
    {
        MaxHits = NumHits;
        ThePlayerWithMaxHits = pIndex;
    }

    gameStat.CheckStatus();
}
```

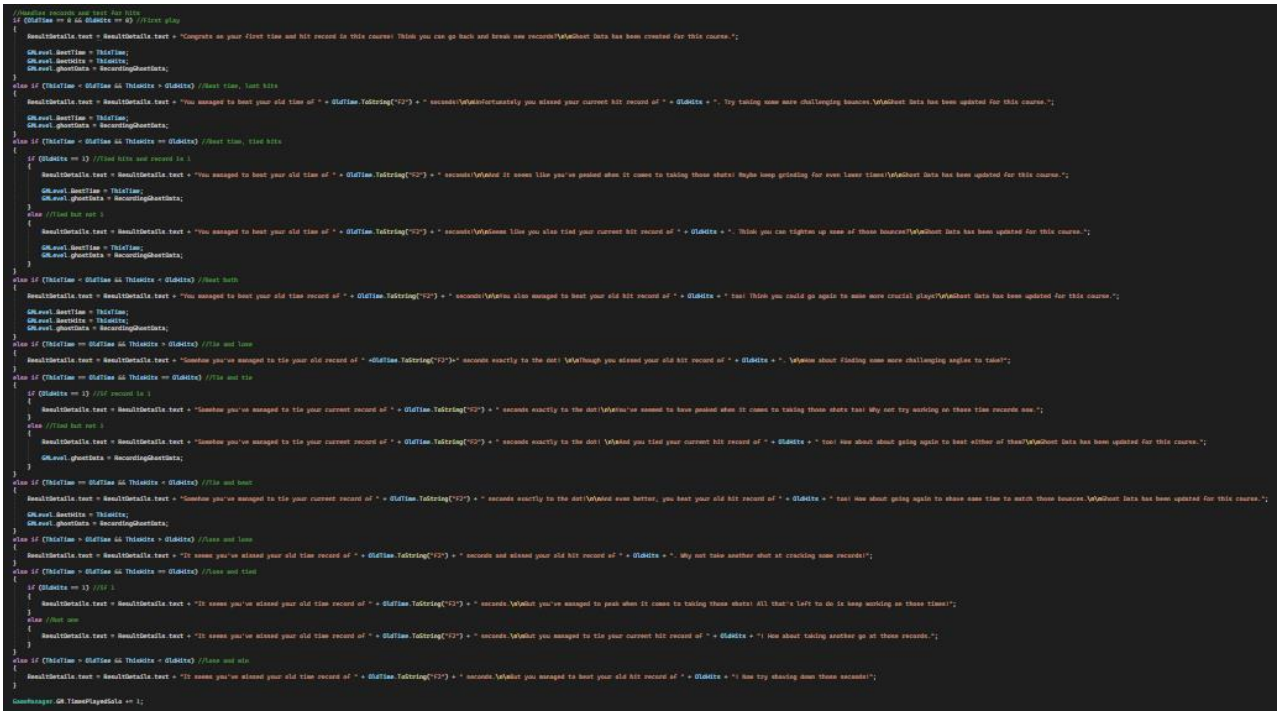
This subroutine will update the list playerStatuses with the player's time, their number of hits, and that they've reached the flag. This subroutine will also return the player's time back to the calling routine so that it can be displayed on their player indicator. Then the game checks whether the player's number of hits is larger than the current largest number and if it is, sets it to that. Finally, the game will check whether the game should end or not in 'CheckStatus()'

```
//Check whether every player has completed yet or not
1 reference
public void CheckStatus()
{
    gameStat.GameOver = true;
    foreach (var item in gameStat.playerStatuses)
    {
        if (item.playerIndex != 99 && item.Completed == false)
        {
            gameStat.GameOver = false;
        }
    }

    if (gameStat.GameOver)
    {
        EndGame();
    }
}
```

In ‘CheckStatus()’, it will assume that the game should be over by setting the flag ‘GameOver’ to true. Then it will check all the players in ‘playerStatuses’ and if there are any players that have not completed, then ‘GameOver’ will be set to false and the game will continue. If all players have ended, then ‘EndGame()’ will be called.

```
//Ends the game when all players have finished
1 reference
public void EndGame()
{
    Debug.Log("All players finished");
    if (SingleMode) //Singleplayer
    {
        var ThisTime = gameStat.playerStatuses[0].Time;
        var ThisHits = gameStat.playerStatuses[0].NumHits;

        var GMLevel = GameManager.GM.LevelData[GMLevelIndex];
        var OldTime = GMLevel.BestTime;
        var OldHits = GMLevel.BestHits;

        ResultDetails.text = "Great job in completing this course in " + ThisHits + " hit/s and " + ThisTime.ToString("F2") + " seconds!\n\n";
```

If the player is in Singleplayer mode, it will first assign variables to be the player’s current time and the player’s best time. Then, depending on whether they’ve beat their record or not, different text will be concatenated to be displayed on the Results Screen. Their records will also be updated accordingly in GameManager ‘LevelData’.



If the game is in Multiplayer instead, then it will just display a single message depending on who managed to get the best time.

```
} else //Multiplayer
{
    ResultDetails.text = "Great job! All players have cleared the course in less than " + gameStat.Timer.ToString("F2") + " seconds! By the way Player " + (ThePlayerWithMaxHits + 1)
      + ", well done with getting it in " + MaxHits + ". Keep working on it!" + "\n\nPlayer 1, feel free to choose whether to Restart Course, Return to Course Select, or just Quit to
        Main Menu.";
    GameManager.GM.TimesPlayedMulti += 1;
}
```

```
if (GameManager.GM.GhostMode)
{
    VsText.SetActive(true);
} else
{
    VsText.SetActive(false);
}

if (InputOne.gameObject.TryGetComponent(out DragAndAimControllerManager manager)) //Changes input mode for Mouse
{
    manager.SetToUI();
}
InputOne.SwitchCurrentActionMap("UI");

ResultsCanvas.SetActive(true); //Show results screen
```

After setting all the text required for the Results Screen, “TryGetComponent” will attempt to find the script ‘DragAndAimControllerManager’ for Player One, which is a special script for the Mouse and Drag Input type. This is because it is unknown whether Player One is using Mouse controls or not, so “TryGetComponent” will not throw an error if the component is not found. If the component is found, then it will fix the player’s input system to work for UIs. After the ‘if’ statement, the ResultsScreen will be shown.

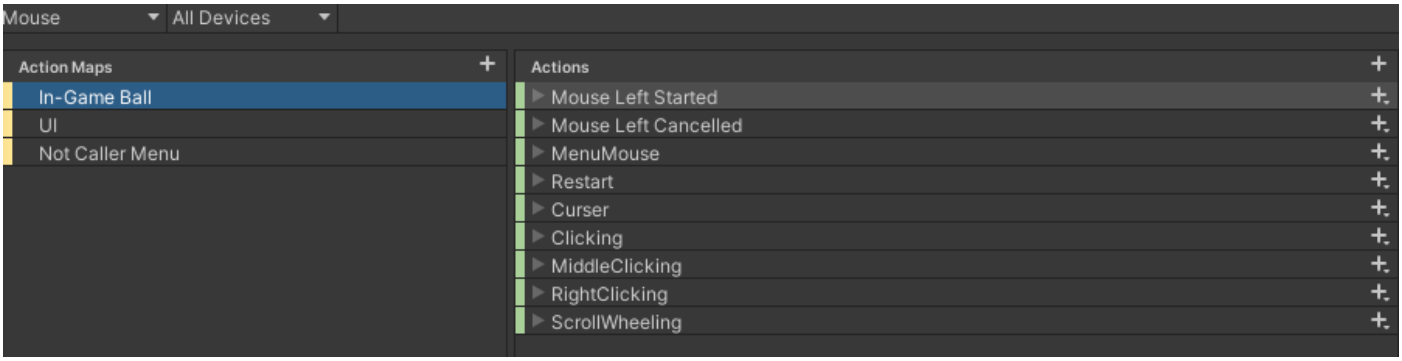
```
//Hijacks player 1 input
POneUISys.playerRoot = ResultsCanvas;
if (InputOne.currentControlScheme != "Mouse")
{
    POneUISys.SetSelectedGameObject(ReturnButton.gameObject);
}
POneUISys.firstSelectedGameObject = ReturnButton.gameObject;

//Disable other player's inputs
foreach (var item in inputs)
{
    if (item.playerIndex != 0)
    {
        item.GetComponent<MultiplayerEventSystem>().enabled = false;
    }
}

//Checks unlockables then saves the game
if (CollectableStatus == 1)
{
    GameManager.GM.LevelData[GMLevelIndex].CollectableGet = 2;
}

GameManager.GM.CheckUnlockables();
GameManager.GM.SavePlayer();
```

Then the player input for Player One will be assigned to the Results Screen so that only they will be able to navigate the UI, which is required in Multiplayer. Meanwhile, other player’s Input Systems will be disabled so that they will not be able to control anything. Both the scroll collectable and ball skin are checked and finally the game is saved.

Player One will then have the option to Restart the Course, Return to Main Menu, or Return to Stage Select.

For Input for Golf, we will first focus on the Mouse and Drag Input Type before Keyboard (or Controller) and Buttons.

Within Unity, actions can be set up so that an action will make a call to a specific subroutine. The following actions are used for in-game Mouse controls.




```
//When mouse is clicked, check if it's the item we want
0 references
public void OnMouseLeftStarted(InputValue value)
{
    if (Physics2D.OverlapPoint(mousePosition) && PauseGame.pM.MenuIsOpen == false && ControllerDisconnectPause.ControlDC.CurrentlyDC == false)
    {
        Collider2D[] results = Physics2D.OverlapPointAll(mousePosition);
        Collider2D highestCollider = GetHighestObject(results);
        targetObject = highestCollider.transform.gameObject;

        offset = targetObject.transform.position - mousePosition;
    }
}
```

When the player holds down the Left Mouse button, ‘OnMouseLeftStarted’ is called to check whether the mouse is being clicked on top of the player’s ball. If it is, then it will set certain variables to allow the aiming indicator to begin rotating.

```
//Every frame, check the position of the mouse and the movement of the ball
Unity Message | 0 references
private void Update()
{
    if (InMotion == true && BallPhysics.velocity.magnitude < 0.005f && !BallMoveScript.FlagHitYet && !BallMoveScript.CurrentlyDead)
    {
        TurnOnThings();
    }

    mousePosition = Camera.main.ScreenToWorldPoint(Mouse.current.position.ReadValue());

    //If the selected object is the one we want, adjust direction and power
    if (ClickableObject.gameObject == targetObject)
    {
        TheObjectWeWantToMove.transform.position = mousePosition + offset;

        XDist = TheObjectWeWantToMove.transform.position.x - ClickableObject.transform.position.x;
        YDist = TheObjectWeWantToMove.transform.position.y - ClickableObject.transform.position.y;

        Vector3 CombDir = new Vector3(ClickableObject.transform.position.x - XDist, ClickableObject.transform.position.y - YDist,
          TheObjectShowingDirection.transform.position.z);

        TheObjectShowingDirection.transform.position = CombDir;

        AngleOfAim = Mathf.Atan2(-YDist, -XDist) * Mathf.Rad2Deg;
        TheActualArrow.transform.rotation = Quaternion.Euler(0, 0, AngleOfAim);

        MaskScaleX = Vector3.Distance(TheObjectWeWantToMove.transform.position, ClickableObject.transform.position) * 1.35f;
        MaskScaleX = Mathf.Clamp(MaskScaleX, 0, 3);
        TheMask.transform.localScale = new Vector3(MaskScaleX, TheMask.transform.localScale.y);
    }
}
```

The rotation of the arrow and the power bar is adjusted through the module's 'Update()' method. If the object that was selected is the same object that we need to select, then it will take the X and Y distance of the player's cursor to the ball and calculate an angle for the direction, as well as the strength of the power for the shot using a couple of maths functions.

```
//When mouse is released and it is the object we want, shoot the ball
0 references
public void OnMouseLeftCancelled()
{
    if (ClickableObject.gameObject == targetObject)
    {
        targetObject = null;
        if (MaskScaleX <= 1)
        {
            Debug.Log("Less than 1");
        }
        else if (MaskScaleX > 1)
        {
            BallMoveScript.ReceiveBallInfo(MaskScaleX, AngleOfAim);

            TurnThingsOff();
        }
    }
}
```

Once the Left Mouse has been released, the subroutine 'OnMouseLeftCancelled()' will check whether the intended object was selected in the first place. If it was, then the targetObject will be reset and it will check whether the power of the shot is valid or not. If the power is less than 1, nothing will happen and the player can re-aim their shot. If it's greater than 1, then it will send the power and direction to 'MoveBallScript' and 'ReceiveBallInfo(power, angle)' as well as disabling any other inputs to the ball while it moves.

Before we discuss about 'MoveBallScript', let us look at the code that controls the arrow and power for Button controls. There are a different set of actions for Button controls that are used for Keyboard and Controller. The script that handles Button controls is called 'AltAim'


```
//Input System Magic
0 references
public void OnAiming(InputValue value)
{
    AimingVal = value.Get<float>();
}

0 references
public void OnPower(InputValue value)
{
    PowerVal = value.Get<float>();
}
```

‘OnAiming’ receives input from either the Arrow Keys Left/Right or Left Stick Left/Right. As it is only on a single axis, a float ‘AimingVal’ is used to hold this value. Similarly, ‘OnPower’ receives input from Arrow Keys Up/Down or Right Stick Up/Down and the float ‘PowerVal’ will hold this value.

```
//Rest of the Script
Unity Message | 0 references
private void Update()
{
    //When the ball has stopped moving, enable things again
    if (InMotion && BallPhysics.velocity.magnitude < 0.005f && !ScriptToMoveTheBall.FlagHitYet && !ScriptToMoveTheBall.CurrentlyDead)
    {
        TurnThingsOn();
    }

    if (InMotion == false && PlayOK == true) //As long as the ball isn't already moving or the game is paused
    {
        ArrowOutline.transform.Rotate(0, 0, -AimingVal * RotMultiplier * (AimingSensitivity / 4) * Time.deltaTime);

        ScaleX = ArrowMask.transform.localScale.x + (PowerVal * MaskScale * Time.deltaTime);
        ScaleX = Mathf.Clamp(ScaleX, 1, 3);
        ArrowMask.transform.localScale = new Vector3(ScaleX, ArrowMask.transform.localScale.y, ArrowMask.transform.localScale.z);
    }

    //Pause needs to be checked at the end of update to prevent ball from launching when closing menu
    if (PauseGame.pM.MenuIsOpen || ControllerDisconnectPause.ControlDC.CurrentlyDC)
    {
        PlayOK = false;
    }
    else
    {
        PlayOK = true;
    }
}
```

Within the ‘Update()’ method, it will check the two float values and if a value is detected, it will rotate the arrow or scale the bar as required.

```
0 references
public void OnShoot(InputValue value)
{
    if (InMotion == false)
    {
        ScriptToMoveTheBall.ReceiveBallInfo(ScaleX, ArrowOutline.transform.eulerAngles.z);
        TurnThingsOff();
    }
}
```

Once the user is ready to shoot, they’ll be able to press the shoot button which will make a call to ‘OnShoot’. As long as the ball isn’t already in motion, it will send the power and angle to ‘ReceiveBallInfo()’ and then disable input to the ball.

Now that the player’s inputs have been received, we are able to begin moving the ball. This is done in the ‘MoveBall’ module with the subroutine ‘ReceiveBallInfo(power, angle)’

```
//Gets the info from Ballthings and converts into angle and speed, then launches the ball
2 references
public void ReceiveBallInfo(float HitStrength, float HitAngle)
{
    audios[2].Play();

    NumHits += 1;
    NumHitsText.text = "Shots taken: " + NumHits;
    if (NumHits >= 1)
    {
        gameObject.layer = 7;
    }

    Debug.Log("Hitstrength: " + HitStrength + " Hitangle: " + HitAngle);
```

At first, the audio clip for hitting the ball will play. Then, the number of hits will be incremented, and if the ball has already left the starting point, then collisions with other balls will be enabled. This is to prevent the balls from getting stuck in each other when they start.

```
    //If the game is singleplayer and or this is the only player, then add list to ghost data
    if (GameManager.GM.SingleMode && playerIndex == 0)
    {
        GameStatus.gameStat.AddGhostData(HitStrength, HitAngle, false);
    }

    LastBallLocation = TheBall.transform.position;

    HitStrength = HitStrength * VelocityMultiplier;

    float XDir = HitStrength * Mathf.Cos(HitAngle * Mathf.Deg2Rad);
    float YDir = HitStrength * Mathf.Sin(HitAngle * Mathf.Deg2Rad);

    TheBall.velocity = new Vector3(XDir, YDir);
}
```

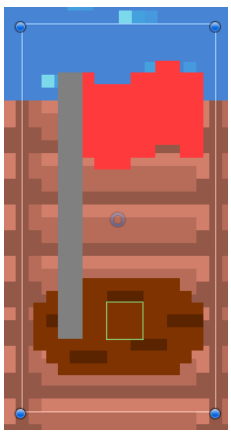
Afterwards, if the player is in Singleplayer and they are Player One, the relevant inputs will be sent to ‘GameStatus’ and ‘AddGhostData(strength, angle, reset)’ so that it can be added to the list of inputs. The ball’s current position is then recorded in case it encounters a hazard before the ball is shot off in the required direction and the required power by manipulating the object’s velocity.

Once the ball has stopped moving by checking its velocity, the corresponding scripts that handle player input will then re-enable input and the user will be allowed to make another shot.

```
if (InMotion == true && BallPhysics.velocity.magnitude < 0.005f && !BallMoveScript.FlagHitYet && !BallMoveScript.CurrentlyDead)
{
    TurnOnThings();
}
```

```
//When the ball has stopped moving, enable things again
if (InMotion && BallPhysics.velocity.magnitude < 0.005f && !ScriptToMoveTheBall.FlagHitYet && !ScriptToMoveTheBall.CurrentlyDead)
{
    TurnThingsOn();
}
```

In order to check whether the ball has already hit the flag, a trigger (indicated by the green square) is set up on top of the flag. Whenever the ball enters a trigger, a subroutine inside ‘MoveBall’ called ‘OnTriggerEnter2D(collision)’ will be called and a number of “tags” will be checked. Tags allow different objects to be differentiated from each other and are used to check what kind of object has entered the trigger.



```
//Handles the ball going into areas it might be in
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.layer == 6) //When the ball hits the water or illegal area
    {
        waterParticle.transform.localPosition = TheBall.transform.localPosition;

        if (collision.CompareTag("TheVoid"))
        {

        }
        else if (collision.CompareTag("Lava"))
        {
            var main = waterParticle.main;
            main.startColor = new Color(245 / 255f, 104 / 255f, 22/ 225f, 255 / 255f);
            audios[0].Play();
            waterParticle.Play();
        }
        else
        {
            var main = waterParticle.main;
            main.startColor = new Color(0, 136 / 255f, 255 / 255f, 255 / 255f);
            audios[0].Play();
            waterParticle.Play();
        }

        CurrentlyDead = true;
        TheBall.velocity = new Vector2(0, 0);
        StartCoroutine(DeathFade());
    }
    else if (collision.CompareTag("Enemy"))
    {
        //audio
        //particle
        audios[3].Play();

        CurrentlyDead = true;
        TheBall.velocity = new Vector2(0, 0);
        StartCoroutine(DeathFade());
    }

    if (collision.CompareTag("Flag")) //When the ball hits the flag
    {
        Debug.Log("Player " + (playerIndex + 1) + " cleared");

        BallHasWon();
    }
}
```

In total, there are three collision checks for whether the ball has encountered level hazards including water, lava, and enemies. If any of these hazards are encountered, the game will play a particle effect, play the corresponding audio of the ball, then reset its position to its last solid location.

```
//Handles when ball has hit flag
1 reference
public void BallHasWon()
{
    audios[1].Play();
    flagParticle.transform.localPosition = TheBall.transform.localPosition;
    flagParticle.Play();

    gameObject.SendMessage("TurnThingsOff");
    FlagHitYet = true;
    TheBall.velocity = Vector2.zero;
    gameObject.layer = 8;
    GameStatus.gameStat.SubmitRecord(playerIndex, NumHits, this);
}
```

However, if the ball hits the flag, then it will call ‘BallHasWon()’. The subroutine would play an audio clip of a ball hitting the goal, play a particle effect, send a message to the input scripts to disable inputs, then send the player’s index, number of hits, and current script to GameStatus ‘SubmitRecord(index, numhits, MoveBall)’ so that it can be processed that a ball has reached the goal. This process has been explained in “*Behind a Game of Golf*”.

While not strictly receiving input from the user, the Ghost ball in Vs. Ghost mode goes through a slightly different process when replaying user inputs.

```
Unity Message | 0 references
private void Start()
{
    //Get required components
    BallRigidbody = GetComponent<Rigidbody2D>();
    spriteRenderer = GetComponent<SpriteRenderer>();

    //Get ghostData and check whether it's ok to play
    ghostData = GameManager.GM.LevelData[GameStatus.gameStat.GMLevelIndex].ghostData;
    if (ghostData.Count == 0)
    {
        Destroy(gameObject); //If data does not exist, delete itself
    }
    else
    {
        //Wait for the go ahead from GameStat
    }
}
```

When the Ghost ball calls its 'Start()' subroutine, it will first take a copy of the user's replay from 'GameManager' and 'LevelData'. If there is no replay to run through, the ball will 'Destroy' itself. But if there is something to play through, the ball will wait for 'GameStatus' to make a call to start the replay.

```
//Once GameStatus has started the game
1 reference
public void StartReplay()
{
    StartCoroutine(IterateSteps());
}
```

```
//Keep looping through the captured inputs
1 reference
IEnumerator IterateSteps()
{
    foreach (var item in ghostData)
    {
        yield return new WaitForSeconds(item.Timing);
        if (item.ResetPos)
        {
            RestartPos();
        } else
        {
            ReceiveBallInfo(item.HitPower, item.HitAngle);
        }
    }
}
```

Once 'GameStatus' makes a call to 'StartReplay()', it will start another IEnumerator 'IterateSteps()' to begin looping through the List of inputs. Before making the first move, it will 'WaitForSeconds(Timing)' based on the actual amount of time that it took for the user to make the shot. This will keep the inputs as accurate to the live run as possible.

If the flag 'ResetPos' is true, then it is known that the user had reset their position and so the Ghost will also replicate that. If not, then 'ReceiveBallInfo(power, angle)' will be called to shoot the ball. While it shares the same name as the subroutine above, it has been modified by removing unnecessary lines of code.

```
//Make moves from inputs
1 reference
public void ReceiveBallInfo(float HitStrength, float HitAngle)
{

    LastBallLocation = transform.position;

    HitStrength = HitStrength * VelocityMultiplier;

    float XDir = HitStrength * Mathf.Cos(HitAngle * Mathf.Deg2Rad);
    float YDir = HitStrength * Mathf.Sin(HitAngle * Mathf.Deg2Rad);

    BallRigidbody.velocity = new Vector3(XDir, YDir);
}

//Reset to last stable position
1 reference
public void RestartPos()
{
    BallRigidbody.velocity = Vector2.zero;
    BallRigidbody.gameObject.transform.localPosition = Vector3.zero;
    BallRigidbody.gameObject.transform.localRotation = Quaternion.identity;
}
```

These tweaks to the Ghost ball allows it to be controlled by the game itself, and removes itself from any player input so that the user can remain focused on trying to beat their best time.

The following class is used to store the player's inputs and to replay them to the Ghost. A List of GhostData is used in 'GameManager.LevelData[]' to store these inputs for saving and loading.

```
//Class to hold data about player's ghost movements
[System.Serializable]
4 references
public class GhostData
{
    public float Timing; //Time between last shot
    public float HitPower; //Power of the shot
    public float HitAngle; //Angle of the shot
    public bool ResetPos; //Did the user reset their position
}
```

Logic and Input for Final Boss

To begin a game against the Final Boss, a number of modules are activated so that the Boss and Players are ready to begin. ‘SpawnShip’ is the module that instantiates the Player’s ship so that they are able to participate in the battle. This module works almost identically to ‘SpawnBall’

```
Unity Message | 0 references
private void Awake()
{
    GameManager gameMan = GameManager.GM;

    if (gameMan.SingleMode || gameMan.NumPlayers.Count == 1)
    {
        int InputType = PlayerPrefs.GetInt("InputType", 0);
        int SkinType = gameMan.BallSkin;

        switch (InputType)
        {
            case 0:
                Player = Instantiate(MousePlayerPrefab, SpawnLocs[1].position, Quaternion.identity);
                break;

            case 1:
                Player = Instantiate(ButtonPlayerPrefab, SpawnLocs[1].position, Quaternion.identity);
                break;

            default:
                break;
        }

        playerShootProjectile = Player.GetComponentInChildren<PlayerShootProjectile>();
        playerShootProjectile.GolfBall.gameObject.GetComponent<SpriteRenderer>().sprite = gameMan.BallSkins[SkinType];
    }
```

On the ‘Awake()’ method, ‘SpawnShip’ checks whether the game is in Singleplayer mode. If it is, it will only have to spawn one ship with the Player’s Ball skin of choice.

```
else
{
    foreach (var item in gameMan.NumPlayers)
    {
        if (item.PlayerIndex != 99)
        {
            switch (item.ControlType)
            {
                case 0:
                    player = PlayerInput.Instantiate(MousePlayerPrefab, item.PlayerIndex, "Mouse", -1, item.inputDevice);

                    player.transform.position = SpawnLocs[item.PlayerIndex].position;
                    playerShootProjectile = player.gameObject.GetComponentInChildren<PlayerShootProjectile>();
                    playerShootProjectile.GolfBall.gameObject.GetComponent<SpriteRenderer>().sprite = MultiSprites[item.PlayerIndex];
                    player.GetComponentInChildren<PlayerHealth>().GolfGear.GetComponent<SpriteRenderer>().sprite = ShipSprite[item.PlayerIndex];
                    break;

                case 1:
                    player = PlayerInput.Instantiate(ButtonPlayerPrefab, item.PlayerIndex, "Keyboard", -1, item.inputDevice);

                    player.transform.position = SpawnLocs[item.PlayerIndex].position;
                    playerShootProjectile = player.gameObject.GetComponentInChildren<PlayerShootProjectile>();
                    playerShootProjectile.GolfBall.gameObject.GetComponent<SpriteRenderer>().sprite = MultiSprites[item.PlayerIndex];
                    player.GetComponentInChildren<PlayerHealth>().GolfGear.GetComponent<SpriteRenderer>().sprite = ShipSprite[item.PlayerIndex];
                    break;

                case 2:
                    player = PlayerInput.Instantiate(ButtonPlayerPrefab, item.PlayerIndex, "Controller", -1, item.inputDevice);

                    player.transform.position = SpawnLocs[item.PlayerIndex].position;
                    playerShootProjectile = player.gameObject.GetComponentInChildren<PlayerShootProjectile>();
                    playerShootProjectile.GolfBall.gameObject.GetComponent<SpriteRenderer>().sprite = MultiSprites[item.PlayerIndex];
                    player.GetComponentInChildren<PlayerHealth>().GolfGear.GetComponent<SpriteRenderer>().sprite = ShipSprite[item.PlayerIndex];
                    break;

                default:
                    break;
            }
        }
```

If the game is in Multiplayer instead, it will iterate through every connected Player from GameManager’s ‘NumPlayers’ and spawn a ship for their corresponding Input Type, as well as replacing the sprite for their Ball and ship.

The module that manages the status of Players and ends the game is called ‘BossStatus’. In its ‘Start()’ method, it will get all the components that are required to interface with the other modules. Then, like ‘GameStatus’, it will find the Input System for Player One. Finally, it will calculate the Boss’ health based on the number of users playing and send it to the Final Boss before starting the game. As the Boss has a short opening animation, the game is started immediately with a call to ‘BeginGame()’.

```
//When start, grab required components, apply boss HP
Unity Message | 0 references
private void Start()
{
    playerInputs = FindObjectsOfType<PlayerInput>();
    inputManager = GetComponent<PlayerInputManager>();
    bossShooting = FindObjectOfType<BossShooting>();
    BossHealth = bossShooting.GetComponent<PlayerHealth>();
    BossAnimator = bossShooting.GetComponentInParent<Animator>();

    foreach (var item in playerInputs)
    {
        playerStatus[item.playerIndex].playerIndex = item.playerIndex;

        if (item.playerIndex == 0)
        {
            PlayerOneInput = item;
            PlayerOneEvent = item.gameObject.GetComponent<MultiplayerEventSystem>();
        }
    }

    int BossHP = 150 * inputManager.playerCount;
    //Get Boss and dump HP into its max health
    BossHealth.healthBar.SetMaxHealth(BossHP);
    BossHealth.CurrentHealth = BossHP;
    BossHealth.MaxHealth = BossHP;

    BeginGame();
}
```

```
//When the loading screen finishes loading, then start the game
1 reference
public void BeginGame()
{
    foreach (var item in playerInputs)
    {
        item.SwitchCurrentActionMap("In-Game");
    }

    GameStart = true;
    bossShooting.PhaseA = true;
    StartCoroutine(IncrementTimer());
}
```

‘BeginGame()’ will instantly allow input to all Players, toggle the flag for ‘GameStart’ to true, begin the Boss’ first phase, then start the IEnumerator ‘IncrementTimer()’ for the timer.

```
//Timer
1 reference
IEnumerator IncrementTimer()
{
    while (!GameOver)
    {
        if (!ForcePause)
        {
            Timer += Time.deltaTime;
        }
        yield return null;
    }
}
```

As this timer is not visible to the user in this mode, text does not need to be updated. However, this IEnumerator still works rather similarly to GameStatus' timer with the difference of it being paused if 'ForcePause' is enabled. 'ForcePause' is used if the game needs to be paused without changing the TimeScale of the game as adjusting TimeScale would cause the Loading Screens to not transition properly.

Instead of waiting for Players to reach the goal, BossStatus will wait for calls from Players that have lost all their HP. This is done through a call to 'UpdatePlayerStatus(index)' where the game will update that user's status as having 'ZeroHP', then checking whether the game should continue or stop.

```
1 reference
public void UpdatePlayerStatus(int pIndex)
{
    playerStatus[pIndex].ZeroHP = true;

    GameOver = true;
    foreach (var item in playerStatus)
    {
        if (item.playerIndex != 99 && !item.ZeroHP)
        {
            GameOver = false;
        }
    }

    if (GameOver)
    {
        PlayersAllDead();
    }
}
```

After checking every player in 'playerStatus', it will decide whether the game should continue or not. However, if all Players have fallen, then 'PlayersAllDead()' will be called instead.

```
1 reference
public void PlayersAllDead()
{
    GameOver = true;
    bossShooting.StopShooting();
    Debug.Log("Players are dead");

    float BossHPRemaining = (BossHealth.CurrentHealth * 1.0f / BossHealth.MaxHealth) * 100;
    Debug.Log(BossHPRemaining);

    ResultsTitle.text = "Boss Course Failed";
    ResultsText.text = "The boss remains victorious. " + Mathf.RoundToInt(BossHPRemaining) + "% of it's HP is left.\n\n";
```

```
if (GameManager.GM.SingleMode)
{
    //Singleplayer logic
    ResultsText.text += "Why not try again to get your revenge! Or feel free to return to the Level Select and Main Menu.";

    GameManager.GM.TimesPlayedSolo += 1;
}
else
{
    //Multiplayer logic
    ResultsText.text += "Why not try again to get your revenge! Player One, feel free to make the calls. Or feel free to return to the Level Select and Main
      Menu.";

    GameManager.GM.TimesPlayedMulti += 1;
}
```

'AllPlayersDead()' will set the flag 'GameOver' to true indicating that the battle has ended, and the Boss will be called to 'StopShooting()'. The percentage of the Boss' remaining health will be calculated and text will be adjusted as required. Players will also be encouraged to try again to defeat the boss.

```
PlayerOneInput.SwitchCurrentActionMap("Menu");

ResultsScreen.SetActive(true); //Show results screen

//Hijacks player 1 input
PlayerOneEvent.playerRoot = ResultsScreen;
if (PlayerOneInput.currentControlScheme != "Mouse")
{
    PlayerOneEvent.SetSelectedGameObject(ResultsFirstButton.gameObject);
}
PlayerOneEvent.firstSelectedGameObject = ResultsFirstButton.gameObject;

GameManager.GM.CheckUnlockables();
GameManager.GM.SavePlayer();
```

The Results Screen will then be displayed with the results of the battle and Input Systems will be adjusted so that Player One will be able to navigate the UI. Ball Skins are checked and then the game is saved in 'GameManager'.

```
//Once the boss is killed, bring up the Results Screen
1 reference
public void BossDefeated()
{
    GameOver = true;
    Debug.Log("Boss is dead");

    ResultsTitle.text = "Boss Defeated";
    ResultsText.text = "Congratulations on defeating the Boss! You managed to do it in " + Timer.ToString("F2") + " seconds!\n\n";

    if (GameManager.GM.SingleMode)
    {
        //Singleplayer logic
        ResultsText.text += "If you'd like to shave some more time to get a better score, feel free to try again! If not, you can always return to Level Select or Main
           Menu.";

        if (GameManager.GM.LevelData[GMLevelIndex].BestTime == 0) //First clear
        {
            GameManager.GM.LevelData[GMLevelIndex].BestTime = Timer;
            GameManager.GM.LevelData[5].CollectableGet = 2;
        }
        else if (GameManager.GM.LevelData[GMLevelIndex].BestTime > Timer) //Beat best time
        {
            GameManager.GM.LevelData[GMLevelIndex].BestTime = Timer;
        }
        else if (GameManager.GM.LevelData[GMLevelIndex].BestTime <= Timer) //Slower than best time
        {
            //Nothing lol
        }

        GameManager.GM.TimesPlayedSolo += 1;
    }
```

However, if the Boss' HP is depleted, then the Boss will send the call to 'BossDefeated()' which will adjust text accordingly, then similarly brings up the Results Screen as well as fixing all the Input Systems.

```
else
{
    //Multiplayer logic
    ResultsText.text += "If you'd like to shave some more time to get a better score, feel free to try again! If not, you can always return to Level Select or Main
        Menu. Feel free to make the call, Player One.";

    GameManager.GM.TimesPlayedMulti += 1;
}

PlayerOneInput.SwitchCurrentActionMap("Menu");

ResultsScreen.SetActive(true); //Show results screen

//Hijacks player 1 input
PlayerOneEvent.playerRoot = ResultsScreen;
if (PlayerOneInput.currentControlScheme != "Mouse")
{
    PlayerOneEvent.SetSelectedGameObject(ResultsFirstButton.gameObject);
}
PlayerOneEvent.firstSelectedGameObject = ResultsFirstButton.gameObject;

//Disable other player's inputs
foreach (var item in playerInputs)
{
    if (item.playerIndex != 0)
    {
        item.GetComponent<MultiplayerEventSystem>().enabled = false;
    }
}

GameManager.GM.CheckUnlockables();
GameManager.GM.SavePlayer();
```

While the battle is on-going, the Boss is constantly shooting projectiles towards the Player. The Boss' attack patterns are handled by 'BossShooting'. Once the call is made from 'BossStatus' to begin, another call is made to the IEnumerator 'ShootProjectile()' which will constantly be active until a call is made to 'StopAllCoroutines()' and pause shooting.

```
1 reference
public void StartShooting()
{
    StartCoroutine(ShootProjectile());
    Debug.Log("Shooting startd");
}

2 references
public void StopShooting()
{
    StopAllCoroutines();
    if (warning != null)
    {
        Destroy(warning, 0.02f);
    }
    Debug.Log("Shooting stopp");
}
```

'ShootProjectile()' is constantly looping around the same routines of code and will only change its attack pattern if specific flags are toggled.

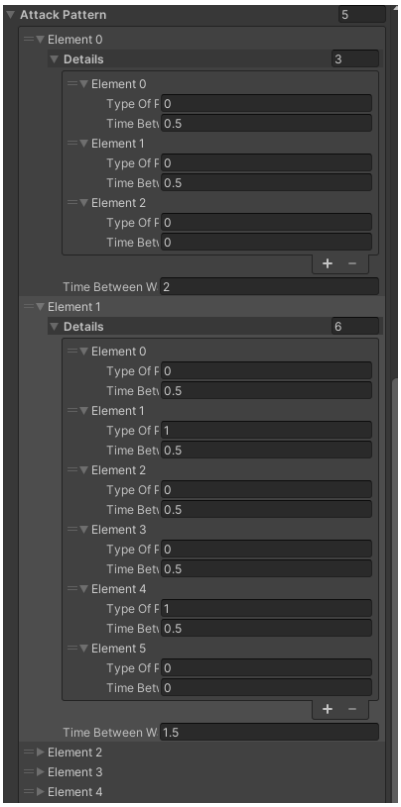
```
2 references
IEnumerator ShootProjectile()
{
    while (true)
    {
        if (BossStatus.bossStat.ForcePause)
        {
            yield return new WaitUntil(() => BossStatus.bossStat.ForcePause == false);
        }

        if (PhaseA) //Phase A Attacks
        {
            yield return new WaitForSeconds(attackPattern[0].TimeBetweenWaves);

            foreach (var item in attackPattern[0].details)
            {
                if (BossStatus.bossStat.ForcePause)
                {
                    yield return new WaitUntil(() => BossStatus.bossStat.ForcePause == false);
                }
                switch (item.TypeOfProjectile)
                {
                    case 0:
                        Instantiate(EnemyBullet, transform.position, Quaternion.identity);
                        AudioManager.instance.PlaySound("IG_bulletboss");
                        yield return new WaitForSeconds(item.TimeBetweenShots);
                        break;

                    case 1:
                        Instantiate(EnemyRocket, transform.position, Quaternion.identity);
                        AudioManager.instance.PlaySound("IG_rocketboss");
                        yield return new WaitForSeconds(item.TimeBetweenShots);
                        break;

                    default:
                        break;
                }
            }
        }
```

These attack phases are based on multidimensional arrays which store the attack that should be shot, as well as the time between attacks. This allows for a range of flexibility as attack patterns can be changed without having to adjust the source code.

The array ‘details’ from class ‘PhaseDetails’ holds information about each attack, such as what attack should be used and the time between attacks.

Then the array ‘attackPattern’ from class ‘AttackPattern’ holds information about each phase, such as how many phases there actually are, and the pattern for each phase.

```
//The attacks in the phases
[System.Serializable]
1 reference
public class PhaseDetails
{
    public int TypeOfProjectile;
    public float TimeBetweenShots;
}

//The different phases
[System.Serializable]
1 reference
public class AttackPattern
{
    public PhaseDetails[] details;
    public float TimeBetweenWaves;
}
```



In order to switch states, ‘StateMachineBehaviour’ scripts are used to track the Boss’ health and to advance the Boss’ phases. Whenever a Phase begins, the Boss becomes vulnerable to attacks (after its transition animation finishes) and the Player will be able to start dealing damage. Once the damage has reached a certain point, a flag will be set through the animator to begin the next phase, which will first play the transition animation, before starting the next shooting phase.

```
Unity Script | 0 references
public class Boss_IdleA : StateMachineBehaviour
{
    PlayerHealth healthScript;
    BossPatternsManager patternMan;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    Unity Message | 0 references
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        healthScript = animator.GetComponentInChildren<PlayerHealth>();
        patternMan = animator.GetComponent<BossPatternsManager>();
        healthScript.IFrames = false;
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    Unity Message | 0 references
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (healthScript.CurrentHealth <= (healthScript.MaxHealth * 0.85))
        {
            animator.SetBool("PhaseB", true);
        }
    }

    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
    Unity Message | 0 references
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        healthScript.IFrames = true;
        patternMan.PauseAttacks();
    }
}
```

```
0 references
public void SwitchSpriteB()
{
    spriteRenderer.sprite = spriteB;
    bossShoot.PhaseA = false;
    bossShoot.PhaseB = true;
}

0 references
public void SwitchSpriteC()
{
    spriteRenderer.sprite = spriteC;
    bossShoot.PhaseB = false;
    bossShoot.PhaseC = true;
}

0 references
public void SwitchSpriteD()
{
    spriteRenderer.sprite = spriteD;
    bossShoot.PhaseC = false;
    bossShoot.PhaseD = true;
}
```

After the projectiles have been instantiated by the Player or the Boss, they must be able to fly towards their target in some way. Here, another IEnumerator ‘ShootForwards()’ is used on the projectile itself that will allow the projectile to fly for a certain amount of time before it gets destroyed.

```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    MyY = transform.position.y;
    MyX = transform.position.x;

    StartCoroutine(ShootForwards());
}

1 reference
IEnumerator ShootForwards()
{
    float time = 0;
    float duration = Duration;

    while (time < duration)
    {
        if (BossStatus.bossStat.ForcePause)
        {
            yield return new WaitUntil(() => BossStatus.bossStat.ForcePause == false);
        }

        MyX += -speed * Time.deltaTime;
        transform.position = new Vector3(MyX, MyY, 0);
        time += Time.deltaTime;
        yield return null;
    }

    Destroy(this.gameObject, 0.02f);
}
```

In order for the projectile to know whether it hit a target or not, ‘OnTriggerEnter2D(collision)’ is also used to check the tags of the entering object and to act accordingly. For a Player’s bullet, it will cancel out with the enemy’s bullet, and get destroyed by all other attacks. Meanwhile, the Player’s missile will beat the enemy’s bullet, cause a health pack to drop against the Boss’ missile, and lose against all other attacks.

```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("EnemyProjectileA")) //Bullets
    {
        Destroy(collision.gameObject, 0.02f);

        if (!BeatEnemyProjectileA)
        {
            Destroy(this.gameObject, 0.02f);
        }
    }

    if (collision.CompareTag("EnemyProjectileB")) //Missiles
    {
        if (BeatEnemyProjectileA)
        {
            Instantiate(HealthDropPrefabs, new Vector3(transform.position.x, transform.position.y, -1), Quaternion.identity);
            Destroy(collision.gameObject, 0.02f);
        }

        Destroy(this.gameObject, 0.02f);
    }

    if (collision.CompareTag("EnemyProjectileC")) //Lazers
    {
        Destroy(this.gameObject, 0.02f);
    }

    if (collision.CompareTag("Enemy")) //The boss itself
    {
        collision.gameObject.GetComponent<PlayerHealth>().TakeDamage(DamageGiven);
        Destroy(this.gameObject, 0.02f);
    }
}
```

Each projectile will deliver a different amount of damage. This damage can be changed on a projective-by-projectile basis for easy balancing. However, once the Player or Boss gets hit, their health needs to be appropriately adjusted. The module ‘PlayerHealth()’ will track and update the UI elements that are relevant for that entity’s health.

```
3 references
public void TakeDamage(int damage)
{
    if (!IFrames)
    {
        CurrentHealth -= damage;
        healthBar.SetHealth(CurrentHealth);

        if (CurrentHealth <= 0 && IsPlayer)
        {
            BossStatus.bossStat.UpdatePlayerStatus(pIndex);

            PlayerDead = true;
            animator.SetTrigger("Death");
            smokeParticle.Play();
            return;
        }

        IFrames = true;

        if (IsPlayer)
        {
            sources[1].Play();
        }

        StartCoroutine(VulnCountdown());
    }
}
```

Due to constraints during development, both the Player and the Boss share the same module for health, therefore depending on whether the flag ‘IsPlayer’ is enabled or not will adjust whether certain effects occur or not. For example, once the entity has taken damage, a call is made to ‘TakeDamage(damage)’ to subtract the damage from the health, update the UI, then check whether the health has reached 0 for the Player. If it has, then the call is made to ‘BossStatus’ to update that Player’s entry and the Player will be unable to continue playing.

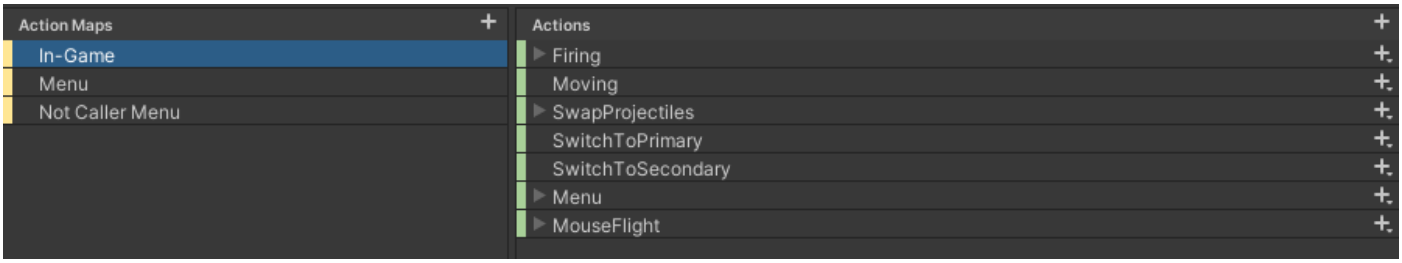
But if the Player still has enough health, they are given a few moments of invulnerability before they will be able to take damage again. This is to prevent the same attack from making the Player take multiple hits of damage, but also as forgiveness for less experienced Players.

For the Boss, the ‘StateMachineBehaviour’ makes the call when the Boss reaches 0 HP. The Boss is also not given any invincibility after it gets hit. Only during phase transitions.

```
// OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks

override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    if (healthScript.CurrentHealth <= (healthScript.MaxHealth * 0.05) && !PhaseE)
    {
        PhaseE = true;
        patternMan.PauseAttacks();
        patternMan.bossShoot.PhaseD = false;
        patternMan.bossShoot.PhaseE = true;
        patternMan.BeginAttacks();
    }

    if (healthScript.CurrentHealth <= 0)
    {
        animator.SetBool("Death", true);
    }
}
```

Whoever depletes their HP first will make the call back to ‘BossStatus’ which will end the game either on the side of the Player or the Boss.

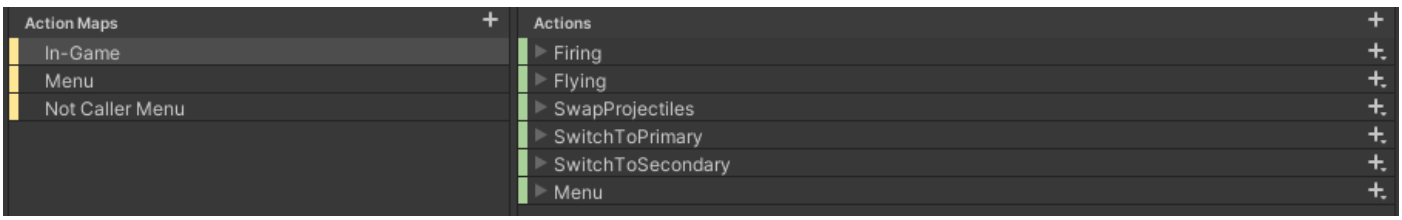
Like the games of golf, the Player Input for Final Boss supports Mouse, Keyboard, or Controller. Firstly, we will look at the actions for Mouse and Drag.



Player movement is handled in the module ‘PlayerMouseMovement’ and is based on the location of the mouse cursor. Specifically, the Y axis of the mouse cursor on the user’s monitor.

```
if (CurrentControls == "Mouse" && !BossControllerDisconnect.BossControlDC.CurrentlyDC && !BossStatus.bossStat.GameOver && !
  BossStatus.bossStat.ForcePause && BossStatus.bossStat.GameStart && !BossPauseGame.bossPause.MenuIsOpen && !pHealth.PlayerDead)
{
    LastYPos = YPos;

    mousepos = Camera.main.ScreenToWorldPoint(Mouse.current.position.ReadValue());
    YPos = Mathf.Clamp(mousepos.y, 5 * -1 + ObjectHeight, 5 - ObjectHeight);

    float diff = (YPos - LastYPos) * Time.deltaTime * 7;

    if (diff > 0)
    {
        diff = Mathf.Clamp(diff, 0, 1);

    } else if (diff < 0)
    {
        diff = Mathf.Clamp(diff, -1, 0);
    }

    float Pos = gameObject.transform.position.y + (diff);
    YPos = Mathf.Clamp(Pos, 5 * -1 + ObjectHeight, 5 - ObjectHeight);
    gameObject.transform.position = new Vector3(transform.position.x, YPos, 0);
}
```

Inside the ‘Update()’ method, as long as the game is not currently paused, the game will get the current Y position of the ship. Then it’ll get the Y position of the mouse and ensure it’s within the bounds of the game. Afterwards, it will get the difference between the ship’s last position and the mouse’s location and restrict those values too. Finally, the ship is moved by that calculated amount. While this gives the ship a bit of a “jelly” effect, it was necessary so that the ship wouldn’t “teleport” if the user paused the game, moved the mouse, then resumed the game. This way, the ship would fly towards the cursor’s new location.



```
0 references
void OnFlying(InputValue value)
{
    MovingValue = value.Get<float>();
}
```

```
else
{
    float Pos = gameObject.transform.position.y + (MovingValue * 10 * Time.deltaTime);
    YPos = Mathf.Clamp(Pos, 5 * -1 + ObjectHeight, 5 - ObjectHeight);
    gameObject.transform.position = new Vector3(transform.position.x, YPos, 0);
}
```

Within the same ‘PlayerMouseMovement’ module is also code to allow for Keyboard and Controller support. The action ‘Flying’ captures either the Keyboard Arrow Keys Up/Down or Controller Left Stick Up/Down. As a single axis, a float ‘MovingValue’ is enough to store this data. Inside the same ‘Update()’ method, the ‘else’ statement calculates the new location of the ship, ensures the ship remains in the bounds of the screen, then sets it to the ship’s new location.

Another module, ‘PlayerShootProjectile’, handles the inputs when it comes to the Player shooting the projectile. This module accounts for all Mouse, Keyboard, and Controller inputs.

```
0 references
void OnFiring()
{
    Holding = !Holding;
}

0 references
void OnSwapProjectiles()
{
    if (CurrentProjectile == 0)
    {
        OnSwitchToSecondary();
    }
    else
    {
        OnSwitchToPrimary();
    }
}
```

‘OnFiring()’ is called whenever the user presses a button corresponding to ‘Firing’. This swaps the flag ‘Holding’ so that if the user is holding the fire button, it will be true. And once they release the button, it’ll set the flag false. ‘OnSwapProjectile()’ is called on ‘SwapProjectile’ and changes the current projectile that the user is using. If they’re using bullets, it’ll switch to missiles. And if they’re using missiles, it’ll switch to bullets.

The subroutines ‘OnSwitchToPrimary()’ and ‘OnSwitchToSecondary()’ can also be called by a user action. Essentially, it swaps the ‘CurrentProjectile’ and swaps the projectile indicator on the Player indicator.

```
2 references
void OnSwitchToPrimary()
{
    BulletImage.gameObject.SetActive(true);
    MissileImage.gameObject.SetActive(false);
    CurrentProjectile = 0;
    WaitTime = 0.3f;
}

1 reference
void OnSwitchToSecondary()
{
    BulletImage.gameObject.SetActive(false);
    MissileImage.gameObject.SetActive(true);
    CurrentProjectile = 1;
    WaitTime = 1f;
}
```

Meanwhile, an always looping IEnumerator ‘HoldingMouse()’ will be keeping track of whether the fire button is active, and if it is it will shoot out the corresponding projectile from ‘CurrentProjectile’. There is also a check to ensure that there are never more than 24 player projectiles on screen at once, as it may cause performance issues and thus be a worse experience for the user.

```
1 reference
IEnumerator HoldingMouse()
{
    while (true) //Always running
    {
        if (Holding) //If player is holding down
        {
            ListOfProjectiles = FindObjectsOfType<PlayerProjectileBehaviour>();

            if (ListOfProjectiles.Length < 24)
            {
                switch (CurrentProjectile)
                {
                    case 0:
                        Instantiate(BulletObject, ProjectileSpawnLocation.position, Quaternion.identity);
                        StartCoroutine(WeaponCooldown(BulletImage, WaitTime));
                        //audios[0].Play();
                        yield return new WaitForSeconds(WaitTime);
                        break;

                    case 1:
                        Instantiate(MissileObject, ProjectileSpawnLocation.position, Quaternion.identity);
                        StartCoroutine(WeaponCooldown(MissileImage, WaitTime));
                        //audios[1].Play();
                        yield return new WaitForSeconds(WaitTime);
                        break;

                    default:
                        yield return null;
                        break;
                }

            }
            else
            {
                yield return null;
            }
```

```
2 references
IEnumerator WeaponCooldown(Slider slider, float duration)
{
    float time = 0;
    while (time < duration)
    {
        slider.value = slider.maxValue - (time / duration);
        time += Time.deltaTime;
        yield return null;
    }
    slider.value = 0f;
}
```

After every shot, a small cooldown is applied to the Player's weapon so that it cannot generate too many projectiles within a certain amount of time. This cooldown is determined by the projectile's 'WaitTime' and a small animation is also played on the player indicator that shows the cooldown decreasing over the projectile's icon.

These couple of subroutines are constantly run until the player runs out of HP, or the Boss is defeated, in which case their input will be disabled or switched to be able to navigate UI.

Logic and Input for Secret Enemy Rush

Due to Secret Enemy Rush being a singleplayer mode, there is no need to instantiate any player characters. The game will begin automatically after the scene is loaded. Thanks to improved knowledge of Object Oriented Programming, different modules that communicate with each other have been created to improve code efficiency.

'FunnyCharMovement' is a class that handles player input, moving the player object, aiming, and shooting the player's projectile.

```
Unity Message | 0 references
private void Start()
{
    pInput = GetComponent<PlayerInput>();
    upgradesScript = GetComponent<PlayerUpgradesScript>();
    mainCam = Camera.main;

    golfSprite.sprite = GameManager.GM.BallSkins[GameManager.GM.BallSkin];

    deathCounterText.text = DeathCount.ToString();
    moneyCounterText.text = MoneyCount.ToString();

    StartCoroutine(Shooting());
}
```

Within its 'Start()' method, it will find all the components required, set the player's sprite to the user's selected skin, set the death counter and money counter to the default, then it will begin the IEnumerator which shoots the projectiles.

```
while (true)
{
    if (HoldingFire)
    {
        switch (CurrentWeapon)
        {
            case 0:
                GameObject theBullet = Instantiate(bullet, arrow.transform.position, arrow.transform.rotation);
                FunnyProjectile funnyProjectile = theBullet.GetComponent<FunnyProjectile>();

                funnyProjectile.damage = damage;
                funnyProjectile.critChance = critRate;
                funnyProjectile.maxHits = maxHits;

                yield return new WaitForSeconds(waitTimeA);
                break;

            case 1:
                for (int i = 0; i < 4; i++)
                {
                    Vector3 finalOffset = arrow.transform.rotation * Vector3ForB[i];

                    GameObject otherBullet = Instantiate(bullet, transform.position + finalOffset, RotationForB[i] * arrow.transform.rotation);
                    FunnyProjectile otherProjectiles = otherBullet.GetComponent<FunnyProjectile>();

                    otherProjectiles.damage = damage;
                    otherProjectiles.critChance = critRate;
                    otherProjectiles.maxHits = maxHits;
                }

                yield return new WaitForSeconds(waitTimeB);
                break;

            default:

                yield return null;
                break;
        }


    }
    else
    {
        yield return null;
    }
}
```

Within this ‘Shooting()’ coroutine, a ‘while’ loop is used to ensure the player will constantly be shooting as holding a ‘shoot’ button for 30 minutes is extremely tiring. Then the ‘switch’ statement will determine which projectile to shoot and calculate the required angle and position to shoot from. Then it will instantiate the projectile with the necessary attributes before waiting a predetermined amount of time to shoot the next projectile

```
Unity Message | 0 references
private void Start()
{
    StartCoroutine(Flying());
}

1 reference
private IEnumerator Flying()
{
    while (time < aliveTime)
    {
        transform.position += transform.right * speed * Time.deltaTime;
        time += Time.deltaTime;
        yield return null;
    }

    Destroy(gameObject, 0.1f);
}
```

The projectiles have their own script called ‘FunnyProjectile’. It begins with an IEnumerator ‘Flying()’ which will keep it flying in a straight line for a specific amount of time. Once time is up or it hits the edge of the map, it will be destroyed.

This script will also handle collisions with enemy objects through the use of a collider trigger. When the bullet passes through an enemy, it will calculate how much damage should be taken and whether the damage is a critical hit. Critical attacks have a low probability of occurring, however they will double the damage that is taken by the enemy.

```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Enemy"))
    {
        DamageCritChance(out int amount, out bool crit);

        collision.GetComponent<EnemyHealth>().TakeDamage(amount, crit);
        collision.GetComponent<EnemyHealth>().TakeKnockback(transform);
    }
    else if (collision.CompareTag("EnemyB"))
    {
        DamageCritChance(out int amount, out bool crit);

        collision.GetComponent<EnemyHealth>().TakeDamage(amount, crit);
        collision.GetComponent<EnemyHealth>().TakeKnockback(transform);
    }

    if (collision.CompareTag("TheVoid"))
    {
        Destroy(gameObject);
    }
}
```

The ‘out’ keyword in ‘out int amount’ is a unique keyword as it allows for the passing of variables by reference instead of by value. This way, both ‘amount’ and ‘crit’ can be passed into and out of the subroutine ‘DamageCritChance(amount, crit)’ without using global variables.

```
2 references
private void DamageCritChance(out int amount, out bool crit)
{
    amount = damage + Random.Range(-2, 2);

    if (Random.value <= critChance)
    {
        amount *= 2;
        crit = true;
    }
    else
    {
        crit = false;
    }
}
```

The ‘amount’ is calculated by taking the ‘damage’ value and adjusting it by plus or minus 2 to add variation to the attacks. A random value is then calculated using Unity’s ‘Random’ class and if this value is less than the ‘critChance’, the ‘amount’ will be multiplied by 2 and a boolean value ‘crit’ will be set to true. Or else, the ‘amount’ is not adjusted. This ‘amount’ and ‘crit’ will then be sent to the corresponding script ‘EnemyHealth’ to take damage.

```
Unity Message | 0 references
private void Start()
{
    internalHealth = enemyHealth;
    cam = Camera.main;
    playerScript = FindObjectOfType<FunnyCharMovement>();

    StartCoroutine(generateRandomTargetPos());
}

Unity Message | 0 references
private void Update()
{
    if (AutoMoveOK)
    {
        transform.position = Vector3.MoveTowards(transform.position, targetLocation + playerScript.transform.position, moveSpeed * Time.deltaTime);
    }
}
```

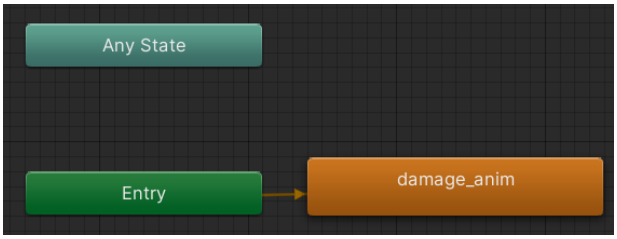
```
1 reference
private IEnumerator generateRandomTargetPos()
{
    while (true)
    {
        Vector2 rndPos = Random.insideUnitCircle * (maxRad - minRad);
        rndPos += rndPos.normalized * minRad;
        targetLocation = rndPos;

        yield return new WaitForSeconds(timeBetweenChangeTarget);
    }
}
```

‘EnemyHealth’ handles both the enemy’s health and movement in this mode. When the script starts, it will begin moving the enemy towards the player. To add extra randomness, the subroutine ‘generateRandomTargetPos()’ will generate a random location that corresponds to an area around the player. This will break up the enemy’s movements once in a while.

```
2 references
public void TakeDamage(int damage, bool crit)
{
    internalHealth -= damage;

    GameObject counter = Instantiate(damageCounter, transform, false);
    counter.GetComponent<Canvas>().worldCamera = cam;
    Text counterText = counter.GetComponentInChildren<Text>();

    counterText.text = damage.ToString();
    if (crit)
    {
        counterText.color = Color.red;
    }
    else
    {
        counterText.color = Color.white;
    }
    counter.transform.localPosition = GetRandomPosition();
```

```
1 reference
private Vector3 GetRandomPosition()
{
    float x = Random.Range(-0.2f, 0.2f);
    float y = Random.Range(0.1f, 0.2f);
    Vector3 pos = new Vector3(x, y, 0);
    return pos;
}
```

When the enemy takes damage, it will subtract its internal health by the damage that was taken. Then, it will instantiate a 'counter' object that will display to the user how much damage the enemy has taken. This is an effective form of feedback to tell the user that the attack did successfully go through. This 'counter' will be updated with the amount of damage taken, and the font colour will be changed depending on whether the attack was a critical attack or not. Finally, it will get a random position on the enemy to show the 'counter'.



The counter has an animation that is played immediately on instantiation. This animation just moves the counter up, then down, before fading out. Once the animation has completed, a 'StateMachineBehaviour' will 'Destroy()' the gameobject to remove it from the scene.

```
if (internalHealth <= 0)
{
    if (Random.value <= spawnRate)
    {
        Instantiate(coinDrop, transform.position, Quaternion.identity);
    }

    Canvas[] canvases = GetComponentsInChildren<Canvas>();
    foreach (var item in canvases)
    {
        Vector3 currentTrans = item.transform.position;
        item.transform.SetParent(null, true);
        item.transform.position = currentTrans;
    }

    playerScript.AddToDeathCounter(1);
    Destroy(gameObject);
}
```

'EnemyHealth' will then check whether the enemy's health has reached 0. If it has, it will randomly generate a value that will determine whether to drop a coin or not. Then it will search for all the

‘Canvas’ within the object and move them out of the object before destroying the enemy. This will ensure that damage counters do not disappear until their animation has completed. Finally, it will add 1 to the player’s death counter and destroy the object.

```
2 references
public void TakeKnockback(Transform bulletPos)
{
    if (co != null)
    {
        return;
    }

    float angle = bulletPos.eulerAngles.z;
    float x = Mathf.Cos(angle * Mathf.Deg2Rad);
    float y = Mathf.Sin(angle * Mathf.Deg2Rad);
    Vector3 posDifference = new Vector3(x, y, 0);

    posDifference = posDifference.normalized * KBAmount;

    co = StartCoroutine(TakingKnockback(posDifference));
}
```

```
1 reference
private IEnumerator TakingKnockback(Vector3 dir)
{
    AutoMoveOK = false;
    float t = 0;
    Vector3 currentPos = transform.position;
    Vector3 targetPos = transform.position + dir;

    while (t < KBTime)
    {
        transform.position = Vector3.Lerp(currentPos, targetPos, t / KBTime);
        t += Time.deltaTime;

        yield return null;
    }

    AutoMoveOK = true;
    co = null;
}
```

‘TakeKnockback(position)’ is also called when an enemy takes damage. This checks whether the enemy has already taken knockback or not. If it has not, it calculates some maths with the bullet’s position and moves the enemy back by a certain amount. This prevents the player from being too overwhelmed during the game. The IEnumerator will move the enemy back over a period of time. Once time is up, the enemy will be able to take knockback again.

```
1 reference
private IEnumerator damageingPlayer(Collider2D collision)
{
    while (true)
    {
        collision.GetComponent<FunnyPlayerHealth>().TakeDamage(damageGiven);

        yield return new WaitForSeconds(TimeBetweenAttack);
    }
}
```

While the enemy is touching the player, it will begin to deal damage to the player over a period of time. An IEnumerator ‘damageingPlayer(collision)’ is used to start a ‘while’ loop that will deliver the appropriate damage to the player. If the enemy stops touching the player, then the IEnumerator is stopped and the player will stop taking damage.

```
1 reference
public void TakeDamage(int amount)
{
    if (currentHealth <= 0)
    {
        return;
    }

    currentHealth -= amount;

    if (currentHealth <= 0)
    {
        currentHealth = 0;
        GetComponent<FunnyCharMovement>().HoldingFire = false;
        funnyMan.CheckTime();
    }

    healthSlider.value = currentHealth;
    healthText.text = currentHealth.ToString() + "/" + maxHealth.ToString();
}
```

The player's health is managed by the 'FunnyPlayerHealth' script. When the player takes damage, it will be passed into the 'TakeDamage(amount)' subroutine. First it will check whether the player's health is already less than 0. If it is, it will just exit the routine. If it isn't, it will subtract the player's health by the damage taken. If the player's health is now less than 0, it will be set to 0, the player is instructed to stop firing, then the Secret Enemy Rush's 'Game Status' will check the status of the game.

In order to effectively manage the mode's Upgrades system, inheritance and encapsulation was used for each player upgrade.

```
Unity Script | 8 references
public class PlayerUpgrades : MonoBehaviour
{
    [field: Header("Base Class")]
    1 reference
    [field: SerializeField] public string UpgradeName { get; private set; }
    24 references
    [field: SerializeField] public int CurrentLevel { get; private set; } = 1;
    3 references
    [field: SerializeField] public string DefaultDescription { get; private set; }

    5 references
    [field: SerializeField] public int[] UpgradeCost { get; private set; }

    2 references
    [field: SerializeField] public MouseHover HoverForDesc { get; private set; }
    8 references
    [field: SerializeField] public MouseHover HoverForAddition { get; private set; }
    1 reference
    [field: SerializeField] public Image UpgradeIndicator { get; private set; }

    protected FunnyCharMovement CharMovement;
    protected FunnyPlayerHealth PlayerHealth;
```

The parent class 'PlayerUpgrades' contains all the methods and attributes that are common between all Upgrades. This includes the upgrade name, its current level, the default description, the cost for each upgrade level, elements for hover, and image for the upgrade counter. References to the player scripts are also included.

The lines '{ get; private set; }' are a special shorthand for encapsulating attributes so that they can only be set by the class itself, but they can be accessed by any class. The keyword 'protected' also ensures that only the parent class and child classes can access those script references.

It was vital to include 'MonoBehaviour' as an inherited class for 'PlayerUpgrades' as it would still allow regular Unity functions to be used and so that it can be attached to an object in the inspector.

Although there are 5 distinct upgrades, all of them work almost identically, so only the 'PickupUpgrade' script will be used as an example in this documentation.

```
Unity Script (1 asset reference) | 0 references
public class PickupUpgrade : PlayerUpgrades
{
    [field: Header("Child Class")]
    9 references
    [field: SerializeField] public float[] PickupRanges { get; private set; }

    Unity Message | 0 references
    private void Start()
    {
        CharMovement.UpdateMagnetRange(PickupRanges[0], CalculatePercentage(PickupRanges[0], PickupRanges[2]));
    }
```

'PickupUpgrade' inherits from 'PlayerUpgrades', which gives it the ability to access all the attributes and methods from 'PlayerUpgrades' as well as all the 'MonoBehaviour' attributes that allow scripts to be used in Unity. The float array 'PickupRanges' stores the different values for each upgrade level.

Within the 'Start()' method, it will set the attributes of the player's "Pickup Range" to the first value inside the array. Similarly for all the other upgrades, its corresponding 'Start()' method will set the default values for each upgrade.

```
3 references
public string CalculatePercentage(int ValueA, int ValueB)
{
    float percentageFloat = (ValueA * 1.0f / ValueB) * 100;
    int percentageInt = Mathf.RoundToInt(percentageFloat);
    string percentageString = percentageInt.ToString() + "%";
    return percentageString;
}

6 references
public string CalculatePercentage(float ValueA, float ValueB)
{
    float percentageFloat = (ValueA / ValueB) * 100;
    int percentageInt = Mathf.RoundToInt(percentageFloat);
    string percentageString = percentageInt.ToString() + "%";
    return percentageString;
}
```

Through the use of polymorphism and overloading, 'CalculatePercentage(A, B)' can either accept two integers, or two floats. Located in the 'PlayerUpgrades' script, it calculates a percentage based on the player's current upgrade, against the player's "middle" upgrade. It will then return the value as a string to be displayed on the HUD.

In order to actually upgrade these abilities, a 'PlayerUpgradesScript' is used to handle the Upgrades menu and process the exchange.

```
Unity Message | 0 references
void Start()
{
    pInput = GetComponent<PlayerInput>();
    movementScript = GetComponent<FunnyCharMovement>();
    playerPause = GetComponent<FunnyPlayerPause>();

    SetupInitialUpgrades();
}

1 reference
private void SetupInitialUpgrades()
{
    foreach (var item in Upgrades)
    {
        item.SetNewIndicator(UpgradeImages[1]);
        item.UpdateDescriptionText();
        item.CalculateStatDiff();
    }
}
```

In the ‘Start()’ method, it will assign all the references required before setting up each upgrade inside the menu with their default indicator, text, and stat difference. Thanks to polymorphism, it will automatically select a child class if it is detected that a method has been overridden.

‘SetNewIndicator(sprite)’ merely replaces the current sprite corresponding to the upgrade.

‘UpdateDescriptionText()’ will update the upgrade’s description text by using its default description while adding the cost at the end.

```
7 references
public virtual void CalculateStatDiff()
{
    print("Parent Class");
}
```

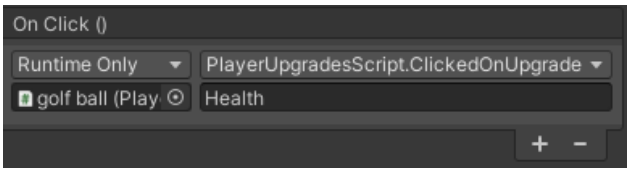
```
3 references
public override void CalculateStatDiff()
{
    float rangeDiff = PickupRanges[CurrentLevel] - PickupRanges[CurrentLevel - 1];
    HoverForAddition.WhatShouldTheTextSay = "+ " + CalculatePercentage(rangeDiff, PickupRanges[2]);
}
```

While ‘PlayerUpgrades’ does have a method for ‘CalculateStatDiff()’, polymorphism allows the program to detect the overriding method and call the child class instead. In the case of ‘PickupUpgrade’, it will calculate the difference between the next and current upgrade before passing it to ‘CalculatePercentage(A, B)’.

The keyword ‘virtual’ indicates to the programming language that the method can be overridden. And the ‘override’ keyword indicates that this is the method that is overriding the inherited method.

```
else
{
    Time.timeScale = 0.1f;
    shopOpened = true;

    normalInputModule.point = gamePoint;
    normalInputModule.move = gameMove;
    normalInputModule.leftClick = gameClick;
    normalInputModule.submit = gameSubmit;

    shopObject.SetActive(true);
    eventSys.SetSelectedGameObject(firstShopItem);
}
```

When the Upgrades menu is opened, the time scale is decreased to create the slow motion effect, and a boolean is set ‘true’ to indicate the menu is open. Some UI attributes are changed so that the player will stop moving the character and be able to navigate in the menu instead. Finally, the menu is opened and the middle upgrade item is selected.



Each button on the menu has been set up so that it will call the same subroutine but with different parameters. This easily allows the script to know what upgrade is being purchased. For example, the Player HP upgrade will pass ‘Health’ into the subroutine, and this string is identical to ‘PlayerUpgrade.UpgradeName’.

```
0 references
public void ClickedOnUpgrade(string UpgradeName)
{
    PlayerUpgrades theUpgrade = Array.Find(Upgrades, u => u.UpgradeName == UpgradeName);
    if (theUpgrade == null)
    {
        if (coroutine != null)
        {
            StopCoroutine(coroutine);
        }

        print("Error. Upgrade " + UpgradeName + " not found");
        coroutine = StartCoroutine(TextForAFewSeconds("Error. Upgrade " + UpgradeName + " not found"));
        return;
    }
```

Within the script, it will use Predicates to find the corresponding upgrade script. The ‘coroutine’ in the following images is an IEnumerator that keeps text on screen for a limited amount of time. If the upgrade cannot be found, an error is printed to the debug log and to the user.

```
if (theUpgrade.CurrentLevel >= theUpgrade.UpgradeCost.Length)
{
    if (coroutine != null)
    {
        StopCoroutine(coroutine);
    }

    print("Upgrade already maxed");
    coroutine = StartCoroutine(TextForAFewSeconds("Upgrade already maxed"));
    return;
}
```

If the upgrade has been found, the subroutine checks whether the upgrade has already reached its final level. If it has, another message is printed to the user.

```
int cash = movementScript.GetCash(); //Current Cash
int required = theUpgrade.UpgradeCost[theUpgrade.CurrentLevel];

if (cash < required)
{
    if (coroutine != null)
    {
        StopCoroutine(coroutine);
    }

    print("Not enough cash. Currently have " + cash.ToString() + ". Require " + required.ToString());
    coroutine = StartCoroutine(TextForAFewSeconds("Not enough cash. Currently have " + cash.ToString() + ". Require " + required.ToString()));
}
```

If the upgrade can still be upgraded, the player's current cash and the upgrade's required cost is compared. If they do not have enough cash, then another message is printed to the user.

```
else
{
    if (coroutine != null)
    {
        StopCoroutine(coroutine);
    }

    print("Successful. Subtracted " + required.ToString());
    coroutine = StartCoroutine(TextForAFewSeconds("Successful. Subtracted " + required.ToString()));

    movementScript.AddToCash(-required); //Subtract amount
    ApplyUpgrade(theUpgrade);
}
```

However, if they have sufficient funds, the transaction will be approved and the cash will be deducted from the player while the upgrade is being applied in the subroutine 'ApplyUpgrade(upgrade)'.

```
1 reference
private void ApplyUpgrade(PlayerUpgrades theUpgrade)
{
    int NewLevel = theUpgrade.SetNewCurrentLevel(theUpgrade.CurrentLevel + 1);
    theUpgrade.SetNewIndicator(UpgradeImages[NewLevel]);

    if (NewLevel >= theUpgrade.UpgradeCost.Length)
    {
        print("Upgrade maxed out");

        theUpgrade.SetNewDesc(theUpgrade.DefaultDescription + " (MAX!)");
        theUpgrade.SetNewAddition("MAX!");
    }
```

‘ApplyUpgrade(upgrade)’ will first check the intended level to upgrade. Then it will set the new indicator above the button through the upgrade’s method. If this is the final level for the upgrade, all of the text will just be set to “MAX!”.

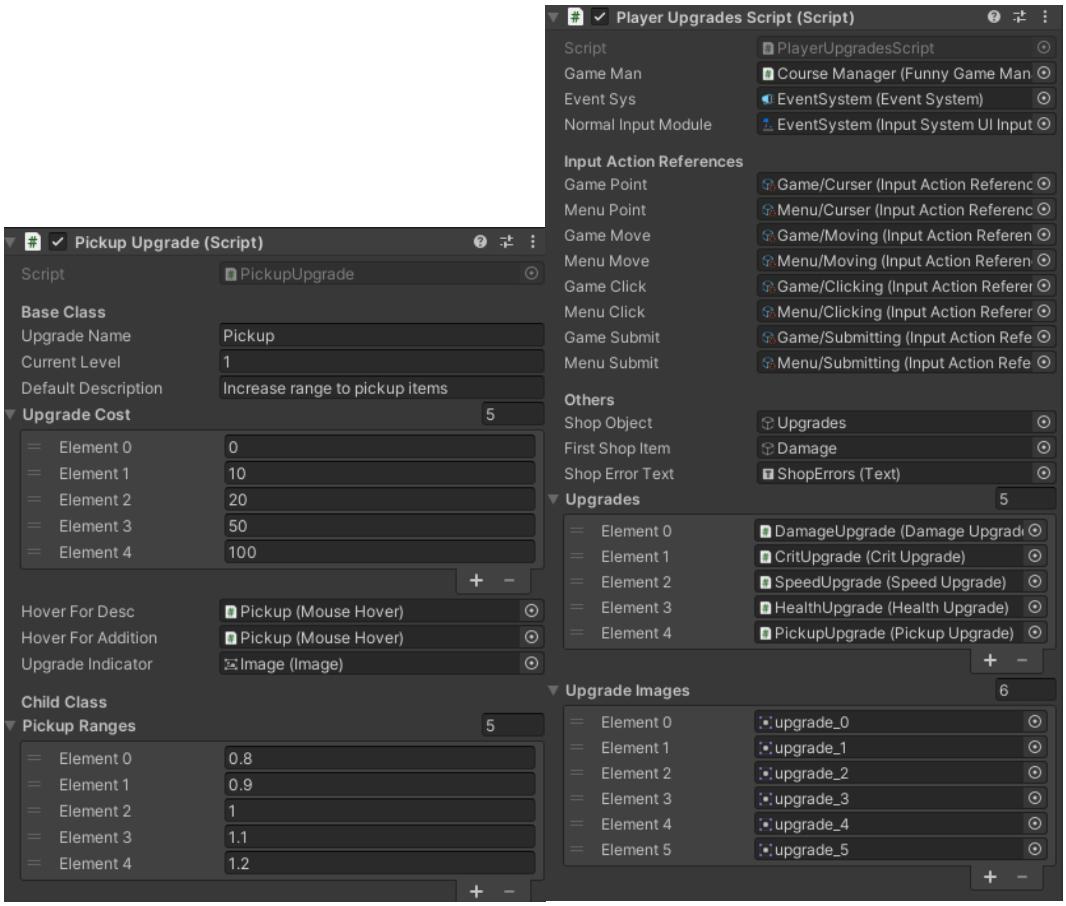
```
else
{
    theUpgrade.SetNewDesc(theUpgrade.DefaultDescription + " (Cost: " + theUpgrade.UpgradeCost[NewLevel] + ")");
    theUpgrade.CalculateStatDiff();
}

theUpgrade.ApplyUpgrade();
```

```
2 references
public override void ApplyUpgrade()
{
    CharMovement.UpdateMagnetRange(PickupRanges[CurrentLevel - 1], CalculatePercentage(PickupRanges[CurrentLevel - 1], PickupRanges[2]));
}
```

Or else, the new description and stat difference will be set via the corresponding script’s methods. Finally, the upgrade’s ‘ApplyUpgrade()’ routine is called to update the internal values which determine the player’s statistics.

```
if (shopOpened)
{
    if (coroutine != null)
    {
        StopCoroutine(coroutine);
    }
    ShopErrorText.text = "";

    Time.timeScale = 1;
    shopOpened = false;
    shopObject.SetActive(false);
    eventSys.SetSelectedGameObject(null);

    normalInputModule.point = menuPoint;
    normalInputModule.move = menuMove;
    normalInputModule.leftClick = menuClick;
    normalInputModule.submit = menuSubmit;


    foreach (var item in Upgrades)
    {
        item.HoverForAddition.ForceDeselect();
    }
}
```

When the menu is closed, the timescale is set back to 1, all the menus are disabled, input actions are reset, and any objects being hovered are disabled.



The above images show an example of one of the upgrade scripts inside the inspector, as well as the 'PlayerUpgradesScript'.

In order to spawn enemies for the different waves, a script 'RandomSpawning' is used to keep track of the number of enemies to spawn, interval between spawning, and how long to spawn a certain wave for.

```
[System.Serializable]
1 reference
public class Spawning
{
    public float RequiredWaitingTime = 1f;
    public int SpawnAtATime = 3;
    public GameObject enemyPrefab;
    public float EndTimeInMinutes;
}

public Spawning[] spawnPattern;

[System.Serializable]
1 reference
public class SpecialSpawning
{
    public float SpawnTimeInMinutes;
    public GameObject EnemyToSpawn;
    public bool HasSpawned;
}

public SpecialSpawning[] specialSpawn;
```

The class ‘Spawning’ handles how many enemies should spawn at a time, time between spawning enemies, the enemy prefab, and when it should stop the current wave. The array ‘spawnPattern’ holds the information for all waves up until the 30 minute deadline is reached.

‘SpecialSpawning’ is a class that holds special enemies that should be spawned once and at a specific time. The array ‘specialSpawn’ holds the information for all special enemies that should be spawned.

```
Unity Message | 0 references
private void Start()
{
    cam = Camera.main;
    BeginSpawning();
}

1 reference
public void BeginSpawning()
{
    StartCoroutine(IncrementTimer());
    StartCoroutine(SpawningCycle());
    StartCoroutine(CheckingSpecialSpawns());
}
```

When the game starts, it will begin incrementing the time, begin the spawning cycle, and begin checking for any special spawns.

```
while (SpawnPossible)
{
    int phase = 0;
    foreach (var item in spawnPattern)
    {
        phase += 1;
        print("Phase " + phase.ToString() + "Started");

        while (item.EndTimeInMinutes * 60 > currentTime)
        {
            for (int i = 0; i < item.SpawnAtATime; i++)
            {
                Instantiate(item.enemyPrefab, GetRandomSpawn(), Quaternion.identity);
            }
            yield return new WaitForSeconds(item.RequiredWaitingTime);
        }
    }

    SpawnPossible = false;
    yield return null;
}
```

‘SpawningCycle()’ will iterate through a ‘while’ loop for as long as there are enemies to spawn. The ‘for’ statement will iterate for as many waves that exist. Then another ‘while’ statement will continuously spawn enemies in the amount that is required. Once the timer reaches the wave’s limit, the ‘for’ statement will move onto the next wave of enemies. Once all waves have been exhausted, it will exit the highest level ‘while’ loop.

```
3 references
private Vector3 GetRandomSpawn()
{
    float x = 0f;
    float y = 0f;

    if (Random.value > 0.5f) //Spawn top or bottom of screen
    {
        x = Random.Range(-5.8f, 5.8f);

        if (Random.value > 0.5f)
        {
            y = Random.Range(2.9f, 3.9f);
        }
        else
        {
            y = Random.Range(-3.9f, -2.9f);
        }
    }
    else //Spawn on the left/right side of the screen
    {
        if (Random.value > 0.5f)
        {
            x = Random.Range(4.8f, 5.8f);
        }
        else
        {
            x = Random.Range(-5.8f, -4.8f);
        }

        y = Random.Range(-3.9f, 3.9f);
    }

    Vector3 pos = new Vector3(x + cam.transform.position.x, y + cam.transform.position.y, 0);
    return pos;
}
```

'GetRandomSpawn()' is a function that will generate a random coordinate on the playing field just outside of the player's view. This creates a sense of randomness to the game and seamlessly allows constant enemies to keep spawning on the field.

```
GameObject[] allEnemies = GameObject.FindGameObjectsWithTag("Enemy");
foreach (var item in allEnemies)
{
    Instantiate(deathParticles, item.transform.position, deathParticles.transform.rotation);
    Destroy(item);
}
```

By the end of the 30 minutes, the player is considered to have beaten the game. But to create a sense of surprise in doing so, all enemies on the field will be destroyed with a particle effect.

```
yield return new WaitForSeconds(3f);

var finalSpawn = spawnPattern[NumOfIndex - 1];
while (true)
{
    for (int i = 0; i < finalSpawn.SpawnAtATime; i++)
    {
        Instantiate(finalSpawn.enemyPrefab, GetRandomSpawn(), Quaternion.identity);
    }
    yield return new WaitForSeconds(finalSpawn.RequiredWaitingTime);
}
```

After 3 seconds, the game will then begin spawning extremely difficult enemies to guarantee the player reaches 0 HP in a quick amount of time. This will call to 'FunnyGameManager' that the game is over.

```
1 reference
private IEnumerator CheckingSpecialSpawns()
{
    if (specialSpawn.Length == 0)
    {
        yield break;
    }

    foreach (var item in specialSpawn)
    {
        while (currentTime < item.SpawnTimeInMinutes * 60f)
        {
            yield return null;
        }

        Instantiate(item.EnemyToSpawn, GetRandomSpawn(), Quaternion.identity);
    }
}
```

'CheckingSpecialSpawns()' meanwhile will spawn a single "special enemy" at the time that is specified. Once all enemies have been spawned, the IEnumerator will end.

When the player reaches 0 HP, either before or after 30 minutes, 'FunnyGameManager' will be called to end the game.

```
1 reference
public void CheckTime()
{
    GameIsActive = false;
    upgradeScript.ForceCloseShop();
    FinalTime = randomSpawning.currentTime;

    pInput.SwitchCurrentActionMap("Menu");

    StartCoroutine(TimingAfterPlayerDies());
}

1 reference
private IEnumerator TimingAfterPlayerDies()
{
    yield return new WaitForSeconds(3f);

    DoEverythingElse();
}
```

'CheckTime()' will indicate the game is over, close the upgrades menu if it hasn't already, record the current time from 'RandomSpawning' and switch the action map to menu.

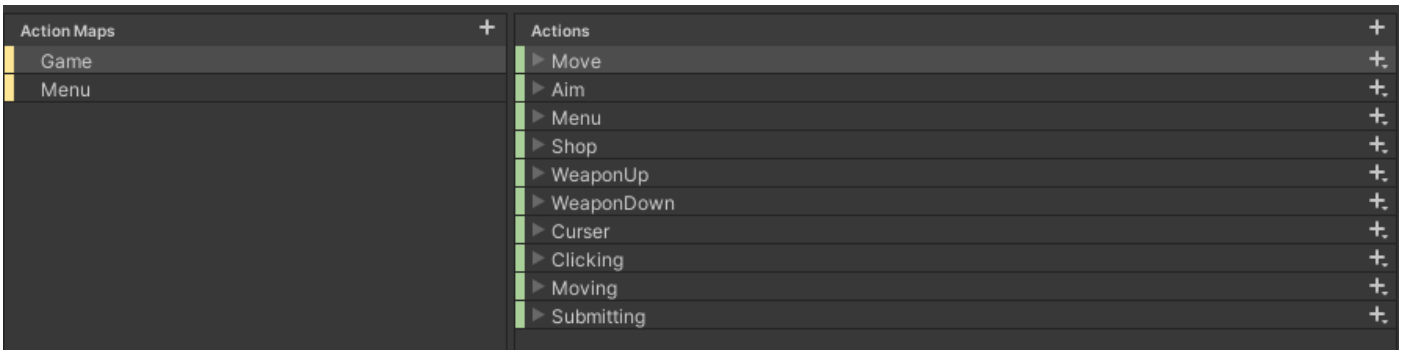
```
1 reference
private void DoEverythingElse()
{
    GameIsActive = false;
    Time.timeScale = 0;

    if (FinalTime >= 30 * 60f)
    {
        PlayerHasWon();
    }
    else
    {
        PlayerHasLost();
    }

    ResultsCanvas.SetActive(true);
    eventSys.firstSelectedGameObject = DefaultButton;
    eventSys.SetSelectedGameObject(DefaultButton);
}
```

Depending on the current time, different text will be displayed on the results screen before it is made active.

```
1 reference
private void PlayerHasWon()
{
    WinnerText.text = "Bonus Stage Cleared";
    TimeLastedText.text = "You survived for 30 minutes!";
    WellDoneText.text = "Thank you for playing this extra bonus mode!";
}

1 reference
private void PlayerHasLost()
{
    int minutes = Mathf.FloorToInt(FinalTime / 60);
    int seconds = Mathf.FloorToInt(FinalTime - minutes * 60);

    WinnerText.text = "Bonus Stage Failed";
    TimeLastedText.text = "You lasted for " + minutes.ToString() + " minutes and " + seconds.ToString() + " seconds.";
    WellDoneText.text = "Why not try again and make it to 30 minutes.";
}
```

The different text is shown as follows. Finally, the player is able to either Return to Main Menu or to Restart the game.

Unlike golf or the Final Boss, there is only one player object for the Secret Enemy Rush. The one input action is able to automatically switch between ‘Mouse and Keyboard’ and ‘Controller’ inputs, and automatically update any relevant text too.



To keep scripts organised, all player movement is located in the ’FunnyCharMovement’ script.

```
0 references
private void OnMove(InputValue value)
{
    moveValue = value.Get<Vector2>();
}

0 references
private void OnAim(InputValue value)
{
    aimValue = value.Get<Vector2>();

    if (pInput.currentControlScheme == "KBMouse")
    {
        aimValue.x -= Screen.width / 2;
        aimValue.y -= Screen.height / 2;
    }
}
```

Since movement is input by the Keyboard’s WASD or the Controller’s Left Stick, it is a 2 directional input and as such, ‘moveValue’ is a Vector2. Meanwhile, aimValue is also a Vector2, but its value will differ depending on the input type. If using a Controller, (0,0) is already centred to the “middle” of the Right Stick. However, for a Mouse, (0,0) is the top left corner of the screen. Therefore, extra code checks if the user is currently using the ‘Mouse and Keyboard’ input and will accomodate for that.

```
Unity Message | 0 references
private void Update()
{
    mainCam.transform.position = new Vector3(transform.position.x, transform.position.y, -10);
    secondCam.transform.position = new Vector3(transform.position.x, transform.position.y, -10);

    if (aimValue != Vector2.zero && !upgradesScript.shopOpened)
    {
        float aimX = aimValue.x;
        float aimY = aimValue.y;

        float radian = Mathf.Atan2(aimY, aimX);
        float angle = Mathf.Rad2Deg * radian;
        arrow.transform.rotation = Quaternion.Euler(0, 0, angle);

        mapPlayer.transform.rotation = Quaternion.Euler(0, 0, angle);

        Vector3 dire = new Vector3(aimX, aimY, 0);
        dire = dire.normalized * 0.35f;
        arrow.transform.localPosition = dire;
    }
}
```

Within the ‘Update()’ function, the two cameras are centred on the player’s object so that it will always be following the player. Then if the upgrades menu is not open and the ‘aimValue’ is not at (0,0), then with some calculations, the direction that the player is aiming in can be calculated and used for firing.

```
Unity Message | 0 references
private void FixedUpdate()
{
    if (upgradesScript.shopOpened) //If the shop is open, keep the user moving in one direction
    {
        if (LerpingVector != null)
        {
            StopCoroutine(LerpingVector);
            LerpingVector = null;
        }

        rb.MovePosition(transform.position + LastDirection);
        ShopJustClosed = true;
    }
```

Since the player is being controlled by a physics object, any code corresponding to player movement is put inside ‘FixedUpdate()’. It differs from a regular ‘Update()’ in that it will only be called every fixed framerate, instead of every frame. This prevents extra calls from being made that could affect the player’s speed. First it will check whether the upgrade menu is open, and if it is, will get the player to continuously keep moving in the same direction.

```
if (ShopJustClosed && moveValue == Vector2.zero) //If the shop has just closed and no movement detected, start automove
{
    LerpingVector = StartCoroutine(LerpVectorToZero());
    ShopJustClosed = false;
}
```

If the shop is not opened, then it will check if the shop was just closed. If it is, it will begin the player’s automatic slide before stopping.

```
else if (LerpingVector == null) //If not lerping then move as normal
{
    ShopJustClosed = false;

    Vector3 tempVect = new Vector3(moveValue.x, moveValue.y, 0);
    tempVect = tempVect * speed * Time.deltaTime;
    rb.MovePosition(transform.position + tempVect);

    LastDirection = tempVect;
}
```

If the player is not being automatically moved, then input will be read from the player and the player will be moved that way.

```
else if (LerpingVector != null && moveValue != Vector2.zero) //If lerping but movement detected, stop lerp
{
    ShopJustClosed = false;

    StopCoroutine(LerpingVector);
    LerpingVector = null;
}
```

Else if the player was being moved automatically, but movement input has been detected, then stop moving automatically to return input to the user.

The IEnumerator ‘LerpVectorToZero()’ just gradually decreases the amount that the player moves per frame until the grace period runs out.

```
1 reference
private IEnumerator LerpVectorToZero()
{
    float duration = 5f;

    for (float t = 0f; t < duration; t+= Time.deltaTime)
    {
        LastDirection = Vector3.Lerp(LastDirection, Vector3.zero, t / duration);
        rb.MovePosition(transform.position + LastDirection);
        yield return null;
    }
    LastDirection = Vector3.zero;
}
```

Transitioning Scenes

In order to hide the jarring effect of moving between Scenes, the module ‘LoadingScreen’ was used to start a transition between Scenes, but to also transition between two different background music. In the module, is an IEnumerator ‘StartLoadMusic(SceneToLoad, timer, oldAudio, newAudio)’ that can pause processing until the new Scene is ready.

Firstly, the routine will check which Scene is going to be loaded and apply the appropriate text and image for it using a ‘switch..case’ statement.

```
1 reference
IEnumerator StartLoadMusic(string SceneToLoad, bool timer, string oldAudio, string newAudio)
{
    BallImg.sprite = GameManager.GM.BallSkins[GameManager.GM.BallSkin];
    loadingSlider.value = 0;

    switch (SceneToLoad)
    {
        case "MainMenu":
            NextStopText.text = "Next Stop: Main Menu";
            NextStopImg.sprite = NextStopSprites[0];
            break;

        case "SampleScene":
            NextStopText.text = "Next Stop: Tutorial";
            NextStopImg.sprite = NextStopSprites[1];
            break;

        case "LevelOne":
            NextStopText.text = "Next Stop: Level One";
            NextStopImg.sprite = NextStopSprites[2];
            break;

        case "LevelTwo":
            NextStopText.text = "Next Stop: Level Two";
            NextStopImg.sprite = NextStopSprites[3];
            break;

        case "LevelThree":
            NextStopText.text = "Next Stop: Level Three";
            NextStopImg.sprite = NextStopSprites[4];
            break;

        case "LevelFour":
            NextStopText.text = "Next Stop: Level Four";
            NextStopImg.sprite = NextStopSprites[5];
            break;

        case "TheFinalBoss":
            NextStopText.text = "Next Stop: Final Boss";
            NextStopImg.sprite = NextStopSprites[6];
            break;

        default:
            NextStopText.text = "Next Stop: Unknown...?";
            NextStopImg.sprite = NextStopSprites[0];
            break;
    }
    NextStopImg.SetNativeSize();
```

```
StartCoroutine(TransitionBGM(oldAudio, 1));

anim.SetTrigger("BeginLoad");
yield return new WaitForSecondsRealtime(WaitTime);
```

After the image and text have been set, another IEnumerator ‘TransitionBGM(oldAudio, duration)’ is used to fade the old audio down to 0. At the same time, the animation trigger ‘BeginLoad’ is toggled so that the transition will begin, and the IEnumerator is asked to ‘WaitForSecondsRealtime(WaitTime)’ which will wait for 1 second without using “scaled time” (which is usually adjusted inside courses). By setting this pause, it gives enough time for the audio to be muted and the transition to be complete before continuing onto the next step.

```
AsyncOperation operation = SceneManager.LoadSceneAsync(SceneToLoad);
while (!operation.isDone)
{
    loadingSlider.value = Mathf.Clamp(operation.progress / 0.9f, 0, 0.94f);
    yield return null;
}
```

The ‘LoadSceneAsync(SceneToLoad)’ instructs Unity to begin loading the new Scene in the background. While this may take longer than loading a scene normally, it prevents the jarring switch between one scene to another. This ‘operation’ can also be tracked, which is done to progress the loading bar on the actual loading screen.

```
StartCoroutine(TransitionBGM(newAudio, 1));

anim.SetTrigger("EndLoad");
yield return new WaitForSecondsRealtime(WaitTime);

if (timer)
{
    yield return new WaitForSecondsRealtime(0.3f);
    GameStatus.gameStat.BeginGame();
}
```

After the scene has successfully loaded, the new audio will fade in at the same time the loading screen transitions out. After the screen has completed its transition, the routine will check whether it needs to begin the timer for a game of golf. If it does, it will wait for a slight fraction before beginning the game. For the Final Boss, the game is instructed to start automatically.

```
4 references
IEnumerator TransitionBGM(string oldAudio, float duration)
{
    print("ping, music change");

    Sound s = Array.Find(AudioManager.instance.sounds, sound => sound.name == oldAudio);
    if (s == null)
    {
        Debug.Log("Uhoh, can't find " + oldAudio + " to play");
        yield break;
    }

    float ogValue = 0;
    float newValue = 0;
    bool WasPlaying = false;
```

In order to fade out the background music, another IEnumerator ‘TransitionBGM(oldAudio, duration)’ was used. This IEnumerator would first find the audio that it has to transition. If it cannot be found, then it would just exit. But if the audio was found, then some variables will be prepared.

```
if (!s.source.isPlaying)
{
    ogValue = 0;
    s.source.volume = 0;

    switch (s.audioPurpose)
    {
        case 0:
            newValue = PlayerPrefs.GetFloat("BGM", 5) / 10;
            AudioManager.instance.CurrentlyPlayingBGM = s.name;
            break;

        case 1:
            newValue = PlayerPrefs.GetFloat("UI", 5) / 10;
            break;

        case 2:
            newValue = PlayerPrefs.GetFloat("InGame", 5) / 10;
            break;

        default:
            break;
    }

    s.source.Play();
}
```

If the audio we are transitioning is not being played, then we want to set its original volume to 0, set the target volume to the user's preference, then start playing the audio.

```
else
{
    WasPlaying = true;
    ogValue = s.source.volume;
    newValue = 0;
}
```

But if the audio is currently playing, then we want to set its current value to its current volume, and set the target value to 0.

```
float time = 0;

while (time < duration)
{
    s.source.volume = Mathf.Lerp(ogValue, newValue, time / duration);
    time += Time.unscaledDeltaTime;
    yield return null;
}
s.source.volume = newValue;

if (WasPlaying)
{
    s.source.Stop();
    s.source.volume = ogValue;
}
```

Finally, over one second, the routine will progressively adjust the volume of the audio until it reaches its target volume. And if the audio was previously playing and is now at volume 0, it will be stopped.

7. Evaluation

Problems that were faced

Throughout development, a number of problems were encountered which required deliberate problem solving and research to be able to solve.

The first major issue that was encountered was my lack of knowledge in using Unity and C#. While I was familiar with Unity as a software, I had never actually used it to design a game before. I also had no prior knowledge with coding in C#, although I had knowledge of coding in other languages. As a result, I devised a plan to create a “test” project first. This way, I could become comfortable with using Unity as a development tool, as well as to learn the syntax and quirks of coding with C#. This test project also allowed me to judge the scope of various ideas including the amount of time and resources it would take to complete. For example, a project based on a “open-world game with an emphasis of completing requests” would have been too large to complete within the limited timeframe. Thanks to overcoming this issue by tackling it head-on, I was able to become more familiar with Unity and C#, as well as improving my understanding on how to solve the problem.

There was an issue I encountered during the implementation of Multiplayer support for the game. Originally, the game was only planned to support a single player. But as development progressed, I realised that I could spend the time and resources in implementing this extra feature, which would greatly increase its complexity, but also the player’s enjoyment of the game. By doing this, I had to install Unity’s new “Input Player System”, which would automatically handle multiple player inputs, but I also had to completely replace the older Input System as they were completely incompatible. Additionally, as this Input System was fairly new, the online documentation was not very satisfactory in instructing how to implement it. As a result, extra resources were spent on learning how to use the new Input System, rectifying any issues that appeared, and ensuring that player input would work for multiple players. Some patience and research were required but in hindsight, this was a great issue to solve as it greatly added value and replayability to the game, and I am extremely proud that it worked out.

While designing the logic for a game of golf, I realised there were some performance issues when the ball hit the flag. As this was encountered early into development, I was somewhat worried that this would cause larger issues further into development as the game became more and more complex. To identify the source of this issue, I ran some debugging tests with stub subroutines and outputted debug messages during vital sections of processing. After a couple of runs, I found that the performance issues came down to the interaction between the ball and the flag. In particular, when the ball enters the flag’s trigger, it would send a call to the flag. The flag would then make another call through the trigger to the ball. The ball would then make a final call back to the flag. This was extremely ineffective as multiple calls were being made back and forth. To fix this issue, it was decided that the trigger should only make a call to the ball. Now, any lines of code that the flag used to handle was now a part of the ball. This instantly cleared up those performance issues and solved the problem.

During development of the save and load system, and specifically adding a checksum to the save file, an issue persisted where trying to read the file resulted in an incorrect checksum and invalid save data. Upon further debugging with output messages, the issue could not be identified. However, after spending a while to calm down, think logically and check the code step-by-step, I realised that I had accidentally serialised the file twice while saving it. Once when converting it to a set of bytes, and again when saving it into the file. This error was due to a lack of experience on my part, so I changed how the save file was written to by using 'WriteAllBytes' instead which instantly solved the problem. This problem also taught me to plan out my modules ahead of time so that issues like these do not appear again.

When working on the UI, I noticed that if a UI element was deselected by the mouse, you would be unable to use the Keyboard or Controller for input. As fluidity between multiple Input Types was a high priority for the main menu, this was a problem that had to be solved. Unfortunately, due to my lack of knowledge about Unity UI, I was stuck on where to start. By this point, I resorted to my usual tactic when I was unsure about how to implement something. Researching past solutions and adapting it to my needs. Thankfully after a while of researching on the internet, I found a forum site which contained a number of lines of code which I was able to understand, and use it as inspiration to develop my own solution. This way, I could solve the problem using my own problem solving while respecting other developer's Intellectual Property rights. This method was regularly used across the software as it would allow me to learn more about Unity while developing my own problem solving skills.

During development of the Secret Enemy Rush mode, I wanted to utilise more Object Oriented Programming attributes such as inheritance, overriding, encapsulation, and polymorphism. As the existing code for the Upgrades system was extremely complicated and ineffective, I planned to remake those modules using the OOP paradigm. However, this would require rethinking the affected modules to effectively implement properly. As a result, I planned the module based on the already existing code and designed the classes for each upgrade around encapsulation and inheritance to fully take advantage of OOP. Splitting up the upgrade subroutines also ensured I could use debugging statements and stubs to test different aspects of the module until it could all be implemented. After rewriting the code and implementing all of the parent and sub classes, the module successfully worked with no major issues, and if required, other upgrades can be added with little work.

What did I learn

Over the course of developing the solution, I have learnt a lot of vital skills related to software design and development. One of the biggest things I have learnt was how to use Unity as a game development software and coding with C#. More specifically, this project has taught me how to deal with in-game physics, player systems, saving, loading, and managing multiple concurrent running modules. Using C# has also increased my knowledge of programming languages that can be used in future projects. This project has also taught me how to use my problem solving skills in identifying an issue, breaking it down into smaller components, and then solving the problem. Within this process, I have learnt how to take my time, think slowly and work logically to solve the problem. Additionally, I have also improved my skills in researching questions and problems that were encountered during development. As a result, I have improved my problem solving skills and will be able to use them in future projects to more efficiently and effectively solve a problem.

How could I improve

In order to allow the solution to be developed within the limited timeframe, the scope of the project was underestimated so that there would be sufficient time to fulfil all of the requirements. Now that the solution has been completed, I would have liked to increase the scope of the game further by either increasing the complexity of the stages, or developing a fully 3D game instead. I could also improve on the design of the game by establishing better requirements such as performance reports or setting benchmarks for improved performance on as many systems as possible. Additionally, I would like to improve the inclusivity of the software. Even though accessibility and ease of use were major requirements for the solution, I wanted to offer the program for other Operating Systems such as Mac and Linux. However, I could not test builds for these systems as I do not own any of them. The solution could also have been improved through the use of more Object-Oriented concepts such as using encapsulation, overrides, and overloads. As these properties of Object-Oriented programming were unknown to me before starting this project, the C# language was not used to the best of its ability throughout the solution, even if it was used in some small part. Ultimately, various improvements could have been made with cleaner UI and more efficient code thanks to the knowledge I have acquired over the time with Unity and C#.

Summary

The original problem was to “create a commercial or educational product for the purpose of a Year 12 Software Design and Development major project”. The requirements state that the solution must be “functional, contain multiple courses, be customisable, have replay value, include multiplayer, and support multiple input types”. After multiple months of researching, designing and developing the software, it can be concluded that the solution fulfils all of the requirements and thus achieves its intended purpose of being a commercial product. The product is highly functional as it performs as expected. There are multiple courses, including a bonus Final Boss and Secret Enemy Rush. The settings screen allows the user to customise their game, adjust their input type, and include accessibility options. And there are various modes that allow the user to replay the game and enjoy it with friends. Therefore, this solution is considered a success for meeting all the requirements.