CS COURSEWORK

CANNONFALL

ANALYSIS

INTRODUCTION

Cannonfall is a 2D platformer meant to be played with a keyboard. Users will control the Cannoneer, a character with a cannon strapped to their back. This cannon allows the user to double-jump (by shooting downwards) and, in doing so, defeat enemies and traverse the treacherous environment. The aim of the game is to overcome obstacles, solve puzzles, and reach the goal – a brand-new power cell for the character's home.

I aim to set the game in a fantasy steampunk style, as I think that meshes well with a cannon and gives a clear aesthetic which is already an established stylistic choice. There will also be checkpoints that correspond to this style, though I am not yet sure of what they will look like.

Cannonfall will have a linear gameplay, with three large levels through which to journey. To spice up progression, I want to add upgrades that allow for a better user experience and more complex sections, like adding a third jump, giving the cannonball an area-of-effect explosion, or otherwise. Due to my hesitancy surrounding balancing and what to add, I intend to use a questionnaire and further research to inform my choices.

The ability to obtain the upgrades may be based on a progression system (i.e. at certain parts of the level gaining a new ability) or using coins or parts dropped by enemies and found in more hidden sections of the platformer.

The enemies will likely be robots to fit with the steampunk aesthetic. They will have simple behaviours (moving back and forth, following the player, following preset paths) and will be able to be defeated by the cannonball released from the double jump.

A potential feature I would like to add is a 'free fire' mode, where the character stops in place and can fire their cannon in any direction. Not only does this add an additional level of complexity to the game, but it also allows me to develop more puzzles when the cannonball doesn't just head directly down. In addition, this means that other enemies can be added, perhaps ones that can launch projectiles of their own, which allows for a crucial element of danger and will hopefully appeal to users.

I have decided to make this game because I am interested in game development in general, and I think that steampunk games have an appealing aesthetic and there aren't many platformers in this genre. The concept is also enjoyable to imagine and develop and gives me a lot of options. In addition, the diversity of the premise allows me to dabble in quite a few different areas (like projectile modelling and level design).

I want to create a game which can be played in short bursts – too many games nowadays require extended periods of time investment to progress. This is unsuitable for many people, who simply want to hop onto a game in their free moments to chill out, have a challenge, and then get back to whatever they were doing. Cannonfall will have long levels, but within them will be frequent checkpoints so that users can play for only a short while and then leave without fearing for their progress.

To further this, Cannonfall should be intuitive and not require a long period of time to understand how to play. This makes the game more accessible to people without demanding a massive time investment.

To create an engaging game, I will need to do research into platformer level design. I plan to use Unity to create a desktop game, coding in C# and using the inbuilt UI editor. A questionnaire will be used to assess controls, requirements, and the UI – data will be gathered from my stakeholders.

RESEARCH

Existing Solution 1: Super Mario Bros. Wonder

The movement style and perspective of the Mario games is a primary inspiration for my platformer. I think that Nintendo's simplistic, formulaic approach to creating a platformer is a good jumping off point from which to add my own approaches.

Super Mario Bros. Wonder - (Nintendo, 2025) – Gameplay

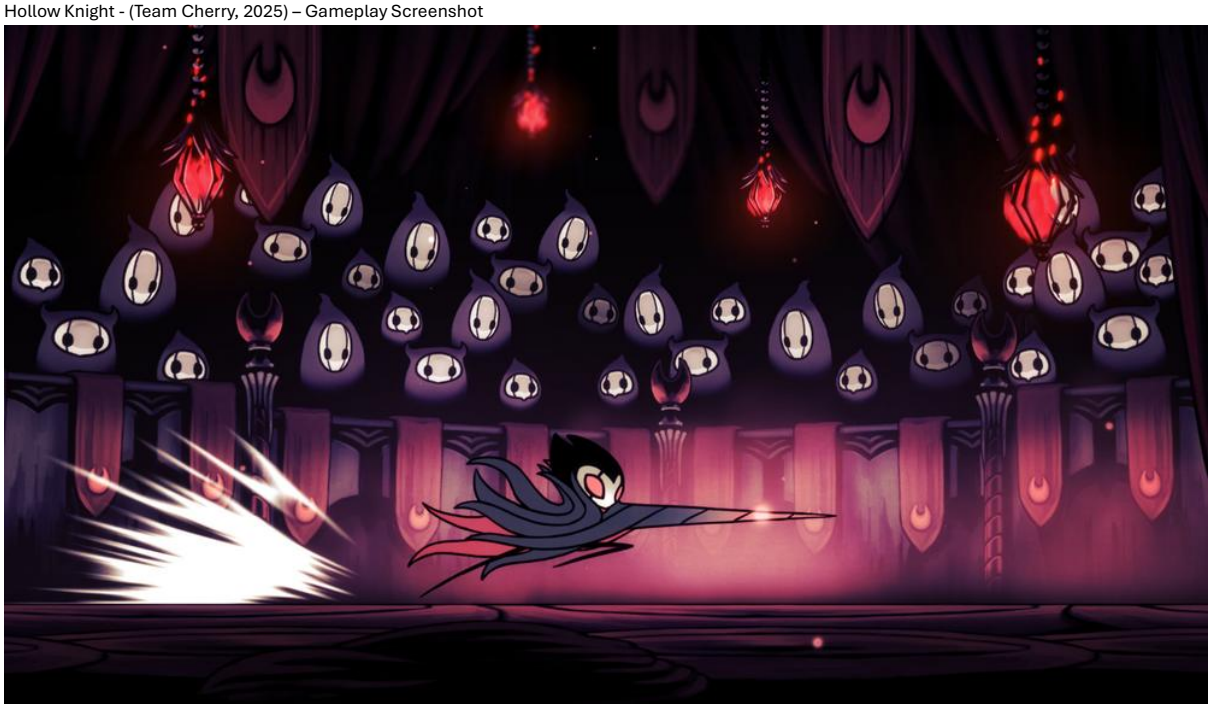
Super Mario Bros. Wonder - (Nintendo, 2025) – Character Roster

Ideas to adopt/adapt into my project	Ideas I would not like to use in my project
Platform simplicity – the platforms are easy to see and distinguish from the background, allowing users to focus on skill in their jumps rather than hoping to see or hit the platforms	**Style & graphics –** the bright colours and rounded style of this game do not fit into the steampunk style I intend to use and lend a more childish feel I wish to avoid.
Enemies – Wonder's enemies have simple pathing and behaviours, which allow users to predict and deal with them when they are encountered. I want to take this into my game but focus on allowing combinations of enemies to increase difficulty.	**Characters –** this game has a large number of playable characters, which would confuse my game (as it revolves around the Cannoneer) and would perhaps lead to seeking a 'meta' character if I implemented different skills/effects for each character. I would rather that be focused on upgrades, leaving players equal at the start of the game.
Token UI – this game has a very clear UI in terms of allowing the user to see how many coins they have collected. My project should contain a similarly obvious counter to remain transparent and allow for a sense of progress, if indeed I include tokens or currency.	**Power-ups –** power-ups in Super Mario Bros. Wonder are ultimately temporary, being used and held by the character. I intend to use permanent upgrades to allow for a sense of progression and perhaps customisation.
Void – the user can fall off the platforms entirely and disappear off the bottom of the screen. I think this makes the game more engaging and lends a level of danger that isn't limited just to the enemies. This would send the user back to the last checkpoint.	**'Death' consequences –** when Mario dies (whether by falling off the map or by an enemy, he resets at the start of the level. In a game that revolves around short burst, multiple checkpoints, and relatively low pressure, this total reset is not suitable for my game.
Scrolling – this game is a scrolling platformer that can also go up and down, rather than just being limited to the horizontal. I wish to adapt this for my game to allow for large levels that aren't just in a straight line, which might get boring.	**Bosses –** this game has a pair of bosses (Bowser and Bowser Jr.), which may be unsuitable for my game because I want the primary focus to be on the platform aspect, not enemies.

Existing Solution 2: Hollow Knight

Hollow Knight's blend of combat and platformer is one which drew my attention as soon as I began to think about making Cannonfall. I believe that this may help me blend the two genres similarly.

Hollow Knight - (Team Cherry, 2025) – Gameplay Screenshot



Hollow Knight - (Team Cherry, 2025) – Troupe Master Grimm (Dive Attack)

Hollow Knight - (Team Cherry, 2025) – Quirrel and the Knight

Ideas to adopt/adapt into my project	Ideas I would not like to use in my project
Combat – Hollow Knight uses a mixture of engaging combat and platform (rather than the simple head-jumping of Mario). I wish to adapt this in my own style, allowing the Cannoneer to battle opponents while (or even for the purpose of) traversing the environment, allowing the use of both cannon-jumping and free-fire shots.	**Health –** Hollow Knight uses a system of 'Masks' to represent health and damage taken (from environment or enemies). Due to the nature of my game as a quick play, and to increase the danger and engagement, I want the character to be vulnerable to all enemies no matter how far the progression.
Environment – the mixture of wide caverns and close spaces is something I definitely want to take	**Atmosphere –** this game has a very solemn, serious atmosphere, which I don't want to take

into my game. I think the use of smaller spaces and enclosed areas removes the more childish aspect seen in Mario and allows for the steampunk aesthetic I want.	forward into my project. The concept is playful and isn't well suited to a graver atmosphere.
Moving platforms – Hollow Knight uses a number of lifts to traverse greater distances in-game. Although I don't necessarily want transport between levels to be done with lifts, they could be useful in changing the aesthetic of certain areas, allowing for greater movement in levels, and can be adapted into moving platforms to increase the difficulty of platforming segments.	**Story –** this game has a heavy emphasis on story and, because of that, boss battles and cutscenes. Due to the platform focus of my game, the story will be simplistic, and the combat meant to enhance rather than distract from it. Due to this, I may use signs or messages to convey story information, but it will be optional to read and will not take away from the platforming itself.
	NPCs – Hollow Knight has a number of non-player characters. This doesn't fit my game at all, as I want to emphasise the isolation of the user's character and remove distractions, which will enhance the dystopian steampunk aesthetic and free the user up from being trapped in lengthy dialogue.

Existing Solution 3: Dead Cells

Dead Cells is a Metroidvania roguelike focused on the singleplayer experience, created in a beautiful pixel-art style. It is this, and the combat-based approach to a platformer, which drew me towards Dead Cells as an inspiration for Cannonfall.

Dead Cells - (Motion Twin, 2025) – The Undying Shores

Dead Cells - (GamePressure, 2025) – Upgrade Screen

Ideas to adopt/adapt into my project	Ideas I would not like to use in my project
Style – the pixel-art style of Dead Cells is very intriguing and fits well with the aesthetic I have in mind. In addition, the darkness and use of sporadic lighting appeals to me as I think that can add an element of danger and links to the steampunk dystopia.	**Elite Enemies –** Dead Cells contains 'elite enemies', which are stronger versions of the regular enemies. I think this is superfluous to my game, as the combat is not the focus and enemies will not necessarily drop loot. By adding elite enemies, I think it would reduce the appeal as a platformer and disappoint players if they beat the enemy with only minor rewards.
Upgrades – after further research, if I decide to do upgrade choices I would like to use a similar system to this game. I think I would prefer to have a small image for each to appeal to users more, but the simplicity of the statistics and the transparency of what you will get is integral to the users understanding what they are obtaining.	**Randomised environment –** this game does not use preset levels and, additionally, has a lot more biomes than the three levels I endeavour to use. I don't think this would work for Cannonfall, as the platformer would have to be designed in order for it to make sense for the minor story and to progress linearly for difficulty.
Biomes – Dead Cells has areas which lead to different events happening, enemies spawning, and similar. Although not as large in scale, I would like to have different areas within my platformer which have different enemies at least, and perhaps modifiers (like darkness or no double jump), as I think this will increase engagement for users. However, they will not be procedurally generated due to the effort that would demand.	**Gear –** gear is integral to most combat-progression games but would serve to only pollute my project. The method of progression, both in combat and in mobility, will be in upgrades – I think adding gear not only skews the game towards combat, but it will also be difficult to implement in a balanced and fair manner and with sufficient variety.
Enemies – enemies have preset behaviours which can be adapted to (for the most part). They also have well-defined spawning points and behaviours, which I need to further consider for my own game. In addition, depending on time and complexity, I may wish to give some enemies abilities to increase variety and challenge.	

STAKEHOLDERS

My target audience is teens and young adults, especially those in the latter years of secondary school and through college (14 – 18 years old). Although it would be suitable for older individuals, as many progress through university they find they have less time for gaming, or less incentive to do so. Cannonfall is aimed towards anyone in that age group who enjoy games and their recreational aspect, without the need for a ranked system or a complex concept. It is particularly suited for those in the period before or exams or at the start of college, where the high workload is stressful but leaves little time for recreational activity. The game will lend the ability to destress in short periods, with the ability to play for as long as they can or want to.

In addition, this project is suited to this demographic because it is a singleplayer experience, meaning that schedules for friends don't need to align for cooperative play and there are no matchmaking wait times. The platformer genre is well-suited to the purpose of recreation – there are no intrinsic competitive elements, and the act of solving puzzles and the success of reaching the next checkpoint can help destress. Cannonfall will be played at home due to its nature as a desktop platformer. It will be used in free time and likely in the evening (after schoolwork has been completed, for example).

My research will be conducted with primarily Y11 and Y12 students (15 – 17 years old) in order to inform user requirements and to conduct testing.

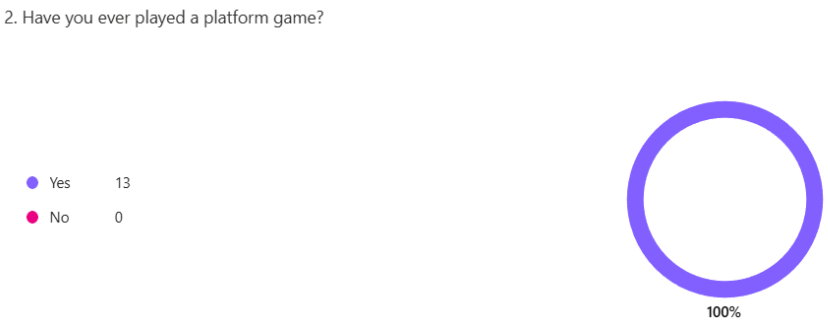
QUESTIONNAIRE

This questionnaire was sent to a small sample of my stakeholders in order to research user requirements and the suitability of my stakeholder group.

Question 1

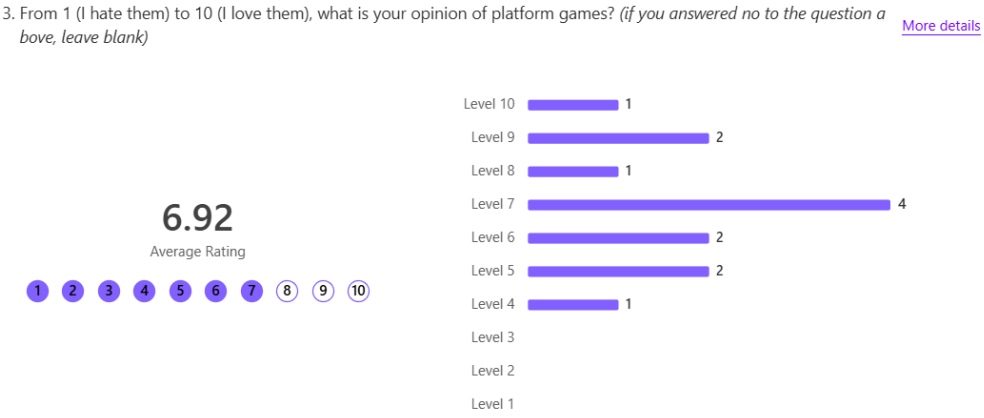
There is a visible skew in the ages of my respondents due to the majority of the sample being taken from colleges in the area. However, this aligns with my stakeholders and shows that the majority of users will indeed be in the targeted age group. The reasons for this are made evident in the 'Stakeholders' section above.

Question 2



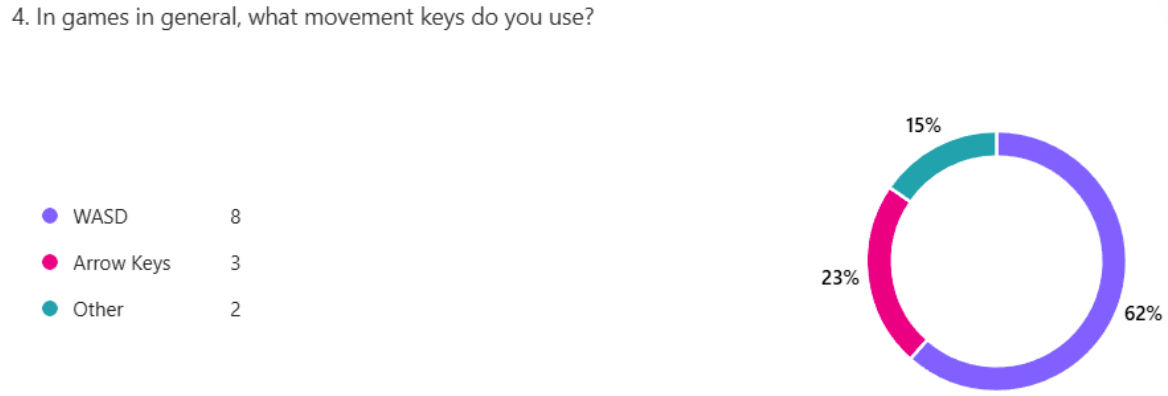
This is an important question because it shows me the general experience of my stakeholders with the game genre. It appears that all of my respondents have some experience with platformers, although I do not plan to rely on this as it is a relatively small sample size. However, this means that I can assume knowledge of certain parts of the game, like not having to explicitly explain the purpose or certain aspects (like not falling, etc.). Due to the simplicity of platformers generally, it makes sense to allow the minority who have not played one before to experiment, rather than dumping a comprehensive tutorial all at once, from the very basics.

Question 3



This question shows a general trend of feeling positive about platform games. This reaffirms that my chosen stakeholders are likely to play my game. This general positivity will create an initial draw to my game, which I then need to supplement with an engaging and unique concept and gameplay. In order to appeal to those with a lower rating, the combat in my game will need to be interesting, and I may add features like puzzles or secret areas to attract those with other interests beyond plain platformers.

Question 4

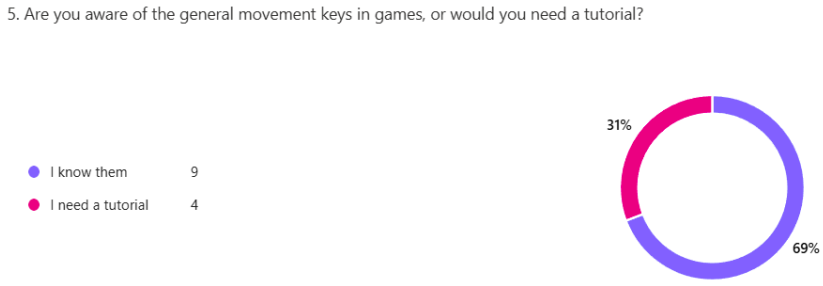


This question shows that the majority of players use the WASD keys for movement, but a significant minority use Arrow Keys. The two 'Others' said *'I'm a touchscreen gamer but if it's keyboard probably WASD'* and *'please allow both'*. The former raises a good point – should I intend to make this game

touchscreen as well? After observing the rest of the responses, none of which mention touchscreen, and deliberating on the time and difficulty of doing so, I decided against it. Creating a touchscreen version would require a significant investment of effort, especially when the respondent in question seems to also use keyboard.

The latter 'Other' respondent requested that both be available and, due to the rather significant minority of Arrow Keys players, I will add functionality that allows both to be used. This will lead to gameplay feeling more natural and appeal to all types of keyboard players – those make up ~92.3% of my respondents, which should represent my stakeholders to a degree of accuracy. By prioritising the feel and inclusivity of the game's controls, Cannonfall will have a greater draw and a higher engagement rate.

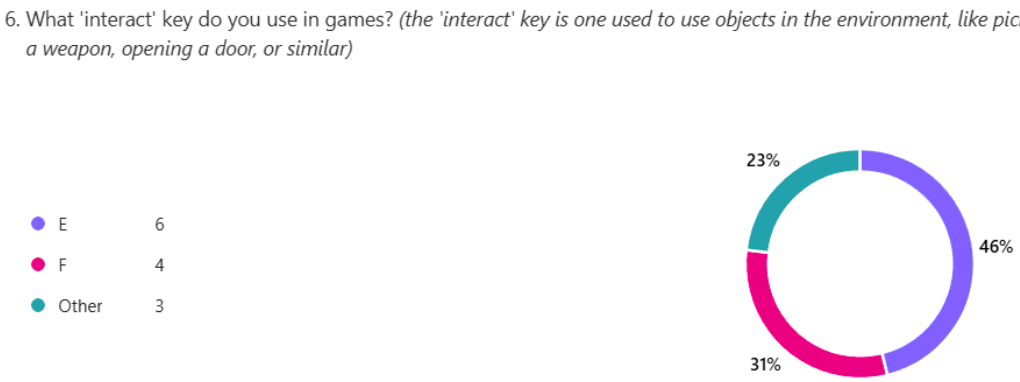
Question 5



A majority of 69% of respondents stated that they were aware of the movement keys generally used in games, showing that most players will be immediately confident in my controls as long as they conform to the norm in video games. However, the remaining 31% would find it difficult to play the game intuitively and have stated that they require a tutorial of some form. This has ensured that I add some form of tutorial, the type of which will be established in a later question.

This ensures that everyone, not just the more veteran gamers, will be able to access and enjoy my game, and makes it certain to me that a tutorial is both justified and necessary.

Question 6



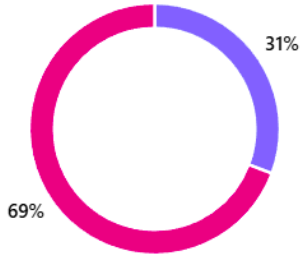
This question was integral to determining what the interact key in my game would be, to make sure that everyone can be catered for where possible. This is a problem I have experienced myself – when playing games, I often find myself having to switch between E and F, which leads to a lag in my gameplay and a feeling of disconnect. Almost half of my respondents said that they used the E key, but the 31% of F-key users is too significant to disregard.

In response to this, I will make both E and F function as interact keys. One of the 'Other' respondents also said *'Z for arrow keys'*, showing that I perhaps disregarded the arrow keys players in the asking of the question, with both F and E being far closer to WASD. For this reason, Z will also work as an interact key. This will cater to the entirety of my respondents and hopefully the near-total majority of my stakeholders, making the game feel natural and immersive.

Question 7

7. What style do you prefer for game tutorials?

Tutorial Level - a full level with the purpose of learning controls, with using that control being the...	4
Signposting - as features become necessary in the actual game, a pop-up or tooltip appears, detailing...	9
None - figure it out as you go	0
Other	0



This question is necessary to find the optimal method of delivering a tutorial. After determining its necessity in question 3, creating the optimal tutorial is necessary to ensure that players have the requisite knowledge to enjoy all stages of the game, but not being dull enough to block access to the rest of the game.

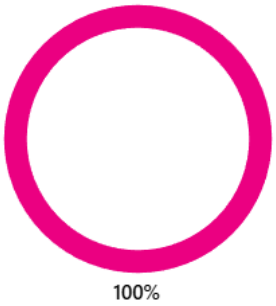
The majority of responses indicated a preference for 'signposting', where features are explained in tooltips as they become apparent. This is the method I plan to include, with a short section of the first level being designated as a tutorial area – this will be very brief and will not have any artificial limits on, due to the majority of stakeholders not enjoying the idea of a tutorial level.

This question also again reaffirms the necessity of a tutorial, as there were no responses saying *none*.

Question 8

8. Do you prefer forced tutorials or optional ones?

Forced	0
Optional	13



My entire sample chose optional tutorials over forced ones, showing an emphasis on being able to choose. Due to this, I plan to add a pop-up at the beginning that asks if they want to see the signposts or not, so users can decide whether they want to figure it out on their own or be told.

Question 9

This question was a text answer question:

Briefly, what are your favourite aspects of a platformer?

I have grouped the results for brevity, in order to address and assess them easily:

- *'challenging gameplay', 'achievement', 'being able to figure out a way past a difficult part'*
    - a number of respondents identified that a key part of a platformer is the difficulty, and the feel of overcoming a challenge. For this reason, I have decided to ensure that the

game has a steep difficulty curve and has puzzle aspects (like targets and moving platforms) to add that extra modicum of difficulty.

- *'fun', 'satisfying', 'uniqueness'*
    - a moderate proportion of responses identified that the game can't take itself too seriously and has to be both fun and satisfying. This means that I need to prioritise the tactile feel of gameplay and ensure that the game is not solely challenging – it is also enjoyable. This is especially important for casual gamers, which a lot of college students especially are (taken from conversations with friends and classmates).

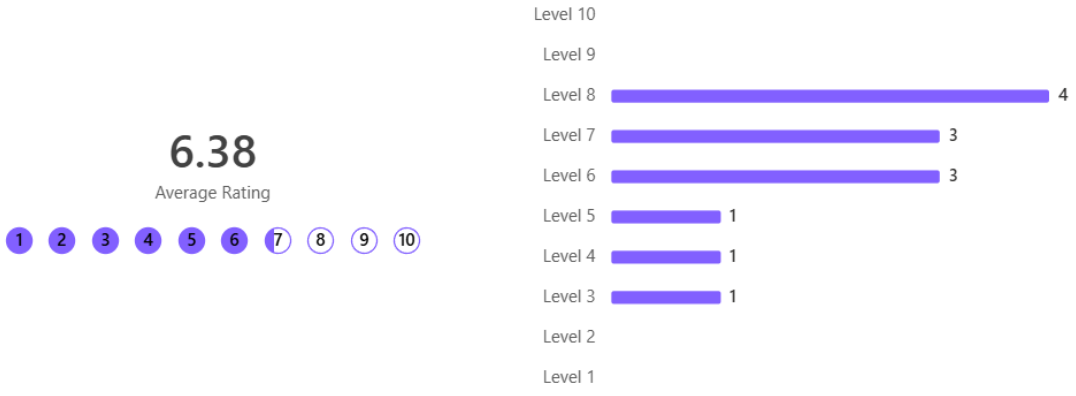
Question 10

10. A platformer requires enemies

~62% of users agree to some extent with this statement. This means that I need to add enemies in order to appeal to this majority, but that the enemies don't have to be the sole focus – not everyone needs them in the game. However, they should still be engaging and interesting, so they don't feel like an add-on just to conform to the genre.

Question 11

11. From 1 (completely ignore) to 10 (kill 'em all), what is your approach to enemies in platform games? More details



There is a skew towards the upper end of this question. I asked this to ascertain the importance of a true combat system to my users. Due to the moderately high average and the consistency of higher responses, it is fairly obvious that a comprehensive way of fighting is necessary. This means that I will create a variety of enemies (so that the more combat-focused players don't get bored) and ensure that combat upgrades appear – for example, explosive cannonballs.

Question 12

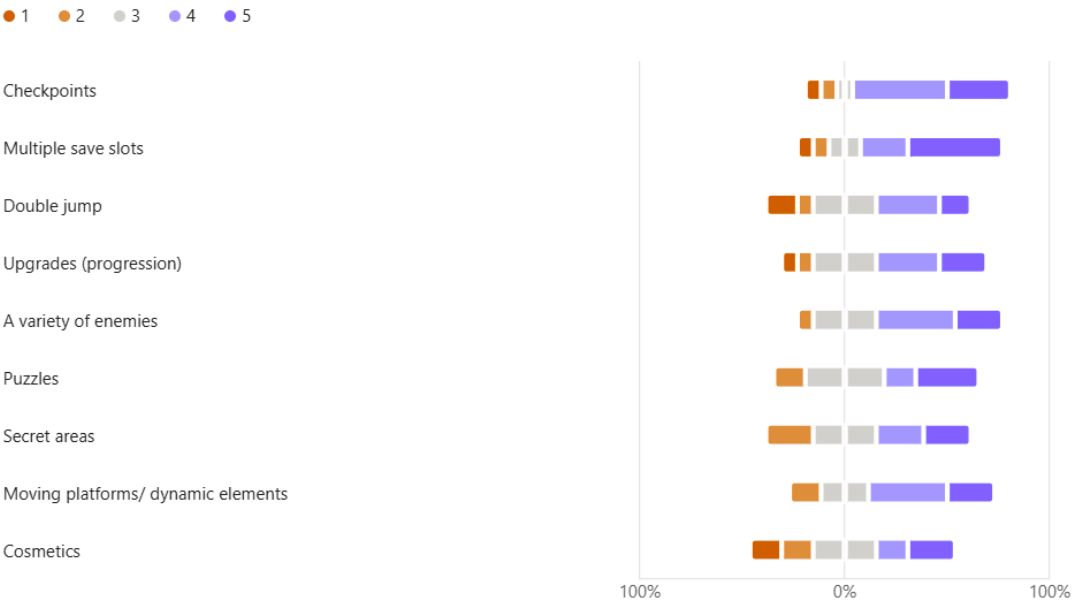
12. Would you prefer a background that moves with you or a scrolling one? *(a scrolling background may repeat infinitely, like a mountain range or clouds, whereas one that moves with you would appear unmoving, like a cavern wall or similar)*

3% of respondents preferred a static background, with 46% preferring scrolling (the Other answer was *'scrolling'*) and the other 31% saying that it didn't impact their experience. This shows me that I have quite a lot of license with how I create the background, as approximately a third do not appear to care, but that I should attempt to create a moving one. I think this dynamic element is better for both immersion and appearance, as shown by its use in a wide variety of platformers, like Hollow Knight.

Question 13

13. Rate the features below on necessity

More details



This is a large question where I got stakeholders to rate prospective features based on import. Checkpoints have been shown to be the most important, meaning that checkpoint functionality is integral and of a very high priority. In addition to this, they should be visible, so that users know when their progress has been saved (a user commented on a prior question that the game should be *'low stakes but still exciting'*, which will be achieved through knowing where you will reset to).

This links well with the high-ranked 'multiple save slots' element, showing that this needs to be prioritised, especially as it has the largest number of 5/5 responses. The responses justify my choice of stakeholders – I identified that an integral part of Cannonfall was going to be short play times, and save slots align with that necessity.

The double jump mechanic, dynamic elements and platforms, and enemy variety are all also high importance, with some of the highest scores. This means that the double jump needs to be fluid and easy to use, as well as simple and obvious. The extra functionality of cannonball shooting will hopefully apply that *'uniqueness'* that was identified by a stakeholder. The dynamic elements and platforms will be high priority also, as they have been identified as necessary by stakeholders, and will help in increasing the difficulty and enjoyment of areas of the levels. The variety of enemies scoring highly validates the analysis of the previous question.

Progression scored lower but will still be included in the 'essential' designation due to being able to make more complex levels due to these upgrades being in line with stakeholder comments about the challenge they desire while platforming.

Puzzles and secret areas scored a little lower, with more negative opinions than prior elements. For this reason, they have been placed in the 'desirable' category – although they are not unwanted by the stakeholders, they are limited in the number of people who enjoy them and are not integral to game functions.

The most spread response was to cosmetics. Although the stakeholder requests would make this a desirable characteristic, the effort involved in making cosmetics (as I am not an art student, with limited abilities in that vein) and the lack of necessity in game functioning make it a very low priority.

Question 14

14. Is there any feature you'd like to see added?

5 Responses

ID ↑	Name	Responses
1	anonymous	points/ prizes (some kind of reward system)
2	anonymous	giant frogs
3	anonymous	Big evil bad guy innit
4	anonymous	Dragons idk
5	anonymous	Extra movement ability such as dash or slide

There were only 5 responses to this question, showing that the majority (~62%) of respondents don't see that anything needs to be added to the game. The request to add *'points/prizes'* is a good one on the surface, but gating areas of levels behind buying the correct upgrade may soft-lock users in certain areas, meaning that purchasing functionality can create issues – if there were no upgrade-specific areas, the upgrades may reduce the difficulty drastically. Cosmetic purchases run up against the justification at the end of Question 13.

Adding a boss, or a *'big evil bad guy'*, would be a prohibitively difficult task for only minor reward. The combat is, while a focus of many users, not integral to all. A boss fight may put off a number of my users, as well as being a major section of work for what is likely to be very little playtime. The extra mechanics I would have to add for a boss mean that it is a low priority, but also one I would like to include.

The addition of both *'giant frogs'* and *'dragons'* is a purely aesthetic choice that doesn't necessarily fit with the aesthetic I have chosen. However, I may take inspiration from *'dragons'* to add flying enemies of some form, and I can see myself hypothetically creating a frog-based enemy to take advantage of its unique movement.

The request for an additional movement ability is a good one, but also one that may be difficult to implement and has only been identified by a single user. For this reason, it will not be allocated to the highest priority but will be ranked highly in the desirable category because it will add a new dimension of play and may make the game more enjoyable.

USER REQUIREMENTS

Cannonfall has a number of necessary features:

- a menu screen from which the user can start the game
- a level screen which shows the current level and the gameplay
- a transition screen that shows the user they have completed a level and moves them to the next
- keyboard controls for the character – WASD or arrow keys for movement and climbing up/down (W and S or up and down arrows respectively), Space to jump and double-jump, and F/E/Z to interact
- three or more levels, each made up of a route of platforms the user can traverse to get from the start to the end position
- enemies which, should they hit the character, will send the user back to the last checkpoint; these enemies should be steampunk robots (e.g. gears, clockwork, hydraulics)
- upgrade pickups at certain points in the levels that increase the functionality of the player character in some way, like adding an extra jump or speeding them up

If I have the time, I would like to add:

- the ability to find and collect a currency system (e.g. 'Cogs') that can be used to purchase cosmetic upgrades
- a 'free fire' mode, where the user is rooted in place but can use the A and D keys to point the cannon in a semicircular arc around the character, and can then fire it in that direction
- a variety of enemies with different behaviours and abilities, like ones that drop from the roof or detonate on proximity
- a co-op mode, where another player can use the arrow keys and other controls to play alongside the first character

I am unlikely to add the following features:

- cutscenes at the beginning and end to immerse the user in the story of the game (that being to get the power cell for the character)
- music and detailed sound effects
- detailed animations

USER REQUIREMENTS TABLE

	Feature		Sub-feature	Explanation	Justification	Importance
1	Start Menu screen	a	New Game button	A button that, when pressed, begins the game from the start	In a game with checkpoints and save data, the ability to start a new game is integral. This allows users to replay the game from the beginning or other users on the same device to start a game without disrupting another's progress.	Essential
		b	Load Game button	A button that, when pressed, allows the user to load a previous save	This button is necessary in order to make use of saved data and checkpoints, to allow users to close the game after playing. All of the games I researched have this ability on the main menu. I intend to follow their lead and add a load menu (see Feature 3).	Essential
		c	Background	A background showing the player character	The use of a background behind the menu screen increases anticipation for the user and makes them more likely to play the game, as well as introducing the character before the game even begins. However, it is not essential to gameplay.	Desirable

		d	Title	A title at the top of the menu screen showing the name of the game	This makes it obvious to the user what game they are playing. The games I researched had very standout titles, which I would like to adopt the user of because I think it increases interest in the game, which may keep the players playing for longer. In addition, it should not be a difficult feature to add.	Essential
		e	Menu navigation	A way to move between choices in menus. W and S or Up and Down Arrows for moving, Enter or F for select	This is necessary to allow the user to actually use the menu, and the variety of keys means that users who play using different key presets will find it natural to use, increasing usability of the menus.	Essential
		f	Quit button	A way to close the game from the menu	This is required to shut down the game entirely, so that the user doesn't need to do it manually. This makes it easier to close the game, promoting the short sessions that Cannonfall is made for, as well as increasing general ease of use.	Essential
2	New Game menu	a	Save select	A choice between the four slots to save it to	This ensures that the user can have four games at once, but also that they can overwrite saves if they want to. Stakeholders identified having multiple save slots as necessary to a good platformer, so this functionality must be assured.	Essential
		b	Overwrite confirmation	If the user chooses to overwrite a save, they are asked to confirm	This makes sure that the user does not accidentally delete their progress by overwriting it with a new game, which will mean players don't quit over losing progress and validates their input.	Essential
		c	Back button	A way to return to the main menu	This feature ensures that the user can back out of a decision, so they don't feel forced to start a new game if they decide they want to load a previous one etc.	Essential
3	Load Menu	a	Load select	A menu with four saves (whether in use or empty), each having some minor details about the save to remember it by	The games I researched had this feature in the load menu – the ability to have multiple games saved at once. This allows for replayability without sacrificing progress and even allows multiple people to play on the same device without overwriting each other's games. Once again, this is an aspect that stakeholders identified as being necessary.	Essential
		b	Back button	A way to return to the main menu	This feature ensures that the user is not trapped in a game they don't want to play if they mis-clicked and allows them to check on their saves and decide to go back and start a new game if they want.	Essential
4	In-game menu	a	Menu	A menu that appears when the 'Esc' key is pressed, pausing the game and allowing the user to quit.	This is an essential feature that allows users to quit without being at a checkpoint. It also means that the user can leave the game running while they do other tasks and it will remain paused, so short sessions can be interspersed with other things even without reaching a checkpoint. The Esc key is used because it is most often used to open the menu in game.	Essential
		b	Continue button	Exits the menu and continues the game. This is both a selectable button and the 'Esc' key	This allows the user to quit the menu and return to the game, to make sure the game can continue if they open the menu. This is also the Esc key because its often used in games to exit menus, so this will make it more natural for users.	Essential
		c	Quit game button	Returns the user to the main menu, but only after a confirmation prompt	This allows the user to the quit the game, which is necessary because the user will play the game in multiple sessions and thus must be able to close it to do other things. However, I have included a confirmation prompt because data is only saved at a checkpoint, so I want to validate the user's choice, so they don't lose progress – this makes them more likely to not play again.	Essential

		d	Menu navigation	*See feature 1e*	*See feature 1e*	Essential
5	Gameplay	a	Levels	At least 3 levels with a course made of various platforms, enemies, and obstacles which the player must traverse	This is the lynchpin of gameplay, essential to being able to play the game itself. The levels will be inspired mostly by the aesthetics of Dead Cells and, to a lesser extent, Hollow Knight. However, the makeup of the level will be a blend of the caverns and large platforms of Dead Cells, the occasional corridor of Hollow Knight, but mostly the quintessential platforms of Mario (in a manner that fits the aesthetic). This is because platforms are eponymous to the genre and adding them immediately draws the interest of the player (by conforming to the nature of the genre) and allows for more dramatic jumps and failures. The course will have a 'void', as in Mario, to add danger to the jumps themselves and test skill.	Essential
		b	Movement controls	Basic controls that allow the user to move forwards and backwards and to jump	The character will move left and right with both the A and D keys or the arrow keys, to cater to either type of platform player (as both of those controls are commonly used, as shown in the questionnaire). Jumping will be done with the space bar (as that is the control used in almost every platformer game and is natural to both control layouts).	Essential
		c	Double jump	While in the air, the user can click Space to jump a second time and launch a cannonball vertically downwards	A double jump ability was shown to be necessary by the stakeholder representatives in my questionnaire. It allows for trickier jumps, a more dynamic experience, and the cannonball gives a method of defeating enemies that feels more realistic than Mario's head-jumping and at the same time keeping the focus on platforming, rather than the more rogue-like Dead Cells and Hollow Knight.	Essential
		d	Dash	To cover a distance immediately (e.g. in order to cross a gap or dodge an enemy) the user can click 'Q' or 'X' to fire in one direction and 'dash' in the other, perhaps becoming intangible while doing so	By request of a stakeholder, and as a new dimension of both movement and combat, this ability is of a middling priority. It would allow for interesting combos, fluid movement, and an easier way to defeat certain enemies. However, this method of combat may be so much easier than the other that it could become the only method, so I may limit to a short-range burst (like grapeshot) to retain balance. Because this was only identified by a single stakeholder it cannot be considered an essential addition, especially as a double jump is a different movement ability that was shown to be necessary.	Desirable
		e	Basic enemies	One or two types of enemy with unique behaviours that, when they make contact with the player, will reset to the last checkpoint	There will be different types of enemy in order to add variety and allow the user to adapt to given scenarios. This will make the game more challenging and more enjoyable. Enemies will die when they are impacted by a cannonball or if they fall into the void. This allows the user to deal with the threat so they can proceed with the platforming without stress. Enemies will be relatively confined to their areas, but some may follow the player a little while to make it more realistic but also unlikely that users will be harried the entire way through. This means that users who don't want to engage in combat will be able to do so, while those who do can fight in enclosed areas without having to worry about missing one and getting overwhelmed in the next area. My stakeholders also have a majority that believe enemies are a necessity, so their existence must be essential.	Essential

		f	Further enemies	More types of enemies (e.g. a spawner or a gunner)	If I have the time to implement them, more enemies will only increase the variety the user will be able to face and increase the mileage of the game (allowing me to extend beyond three levels if I wish). The behaviours will be more radically different, which will increase the challenge level the users will have to face. This is particularly inspired by Dead Cells, where there is a massive variety of enemies based on location – although I am unlikely to reach the same variety, the complexity of having a wide range of enemies makes the game much more interesting and mysterious, increasing player retention. Stakeholders expressed an interest in larger enemies, like 'giant frogs' and 'dragons', so an additional enemy may cater to those tastes.	Desirable
		g	Upgrades	Upgrades that are gained through game progression, like having an extra shot (triple jump), explosive cannonballs, or being faster	As shown in Dead Cells and Hollow Knight, progression is integral in combat-platformers. While I'm leaning further towards platformer than rogue-lite, this progression is still necessary in order to keep the gameplay from feeling stale and allow the introduction of stronger enemies. Stakeholders identified this as a middling priority, but the majority also identified challenge as an integral part of a platformer, and more developed abilities can really push that challenge to another level. Thus, it is essential but fairly low within that category. I intend to use a screen similar to that in Dead Cells, with a very obvious notification on what the powerup does. I like the idea of customising the upgrades, so a choice between two upgrades may be added, but that is desirable rather than essential as a progression of upgrades is more necessary than variety.	Essential
		h	Environmental Hazards	Parts of the level that can kill the player, like spikes or the void	A staple in most platformer games, having parts of the level that can reset the character to the last checkpoint gives some stakes to the parkour – if the user misses a jump, they may be punished with a large reset. This increases engagement by making it more challenging. The use of a 'void', which the character can fall from, is inspired by Super Mario Bros. Wonder and appears in many different platformers. This increases tension in a different manner to something like spikes, by making the space feel more open and therefore the jumps more perilous.	
		i	Level end screen	A short screen that will display over a background of the transition from one level to the next, perhaps a typical 'tips' screen or a little bit of progression of the 'story', such that it exists	This gives a clear ending, allowing users to feel a sense of success, as well as giving them a good time to end a gaming session or to push on near the end. This is dissimilar to what is seen in Mario, as that game has very distinct levels – Cannonfall should have more flowing levels with a more natural transition, to give a sense of reality and to flesh out the location. However, this is not integral to the functioning of the game and so can be sidelined in order to ensure necessary parts are developed fully.	Desirable
		j	'Free fire' mode	The character is, at the click of a button ('Tab' or 'C') stopped in place and the cannon is unslung from their back. They can then use the A and D or left and right	This mode is by no means essential to the functioning of the game. However, it does give an extra dimension of use to the cannon, which is core to the concept of the game, and allows for puzzles and problem solving. In Mario, for example, temporary powerups like Fire Flowers and Boomerangs fill the same niche in allowing an entirely new concept to improve the gameplay	Desirable

				arrows to move their cannon along a semicircular arc around them and fire a cannonball using space in that direction. The cannonball moves under gravity.	experience. It also allows for more complex enemies and interactions with the environment, giving a reasonable counter to long-range enemies and allowing the user to do things like hit targets to open doors and the like. Because of the way the cannonballs will arc and the lack of trajectory line, there will be a learning curve to the free fire mode I think will entertain and immerse the user even further into the game. However, this is a low priority even within the Desirable category due to it being superfluous to aspects identified as core.	
		k	Checkpoints	Easily visible places at which the game will save, allowing the user to quit and load the game from there and marking where the player will return to if they die.	Checkpoints are absolutely integral to my game. As seen in Dead Cells, Hollow Knight, and Super Mario Bros: Wonder, they are a part of the platformer genre. They allow users to feel calmer after completing a particularly difficult section, knowing that they will respawn after it if they fail later. In addition to the active gameplay experience, my game has a primary appeal due to short play times, which is facilitated by having a number of checkpoints where the game can be saved. As shown in the questionnaire, users place a high priority on checkpoints and so they must be given a high priority, and it will help facilitate a challenging but enjoyable game.	Essential
		l	Elevators	Elevators are used to cover large distances in a level without having to transition between levels.	Elevators would primarily be a device to change level aesthetic without ending a level, as ending a level will likely come with a paradigm shift in design or an influential upgrade. This will allow users to feel immersed – appearance of the level won't just shift drastically for no reason, and it gives some variety rather than just moving in a single direction the entire level. I may also use them to finish a level, but they will be visibly distinct. This is inspired by Hollow Knight, in order to lend some dynamic elements to the setting – however, this is not integral to the game.	Desirable
		m	Moving platforms	Moving platforms would be small platforms that move along linear paths	These would add some difficulty and increase the importance of the environment. Inspired by some of the platforms in Super Mario Bros: Wonder, they would test the player's timing and thus increase the challenge level. It would also help in giving a sense of newness to every level, so users don't get bored. My stakeholders have also identified that these are necessary to a platformer, so they are a priority.	Essential
		n	Ladders	Vertical objects that can be ran past but, on interaction, can be climbed up or down	Ladders appear in some form in most platformers, not least as Vines or vertical Poles in Wonder, and simpler ladders in Hollow Knight. They allow for a new dimension of travel and simply enhance the feel of the game as a 'true' platformer. They once again improve the variety and therefore are likely to increase player retention.	Desirable
		o	Targets	Objects that can be shot, which unlocks a linked door or moves a platform	Targets allow an extra challenge to be added to parts of the level which isn't solely based on jumping skill and timing. Respondents to the questionnaire largely identified that a challenge is a key part of a platformer, and approximately half identified that puzzles were high priority. I intend to follow through with this opinion, sprinkling puzzles of variable difficulty throughout the game, but it is not absolutely essential to the gameplay itself.	Desirable
		p	Boss(es)	Enemies with more health, drastically different actions, and	When queried about what features they would like to see added, my stakeholder sample identified larger enemies and a *'big evil bad guy'*, which I take to	Desirable

				a much higher challenge rating. They will be used to gate certain parts of the level (like the ending)	mean boss. Due to clear interest, I would like to add a boss, however the difficulty of such an endeavour limits its implementation slightly – I would need to add an entire new sprite, health mechanics, and different attacks.	
6	Shop			A menu or location in which a currency system (which could be collected from the platforms or dropped by enemies) could be spent to buy skins for the character.	A stakeholder desire is points/prizes, which would then assumably be spent on something in order to give them some purpose. Due to the progression nature of my platformer and the need for all users to have the same functional capabilities, I don't want upgrades to be based on a pure currency system, so all users can fundamentally overcome all challenges. So, cosmetics make the most sense to be purchased. The creation of cosmetics is an issue however, as my artistic ability is limited, and it is not integral to the game so it can only be called desirable.	Limited
7	Local co-op			Because the game already uses two sets of controls (those surrounding WASD and those around the Arrow Keys), local co-op would link one set to each character so that two users can play on the same keyboard.	Local co-op is by no means integral to the functioning or gameplay of the main game, but the added draw of multiplayer with friends is an idea inspired by Mario, which I researched. Many platformer games rely on a cooperative element to increase retention. However, this is a truly last-resort desirable option; although it would not be overly challenging in theory, adding a new character could create difficulty and it is not identified as a key stakeholder request.	Desirable

Limitations

	Limitation	Explanation & Justification
1	Animations	In a true game, fluid and anatomically correct animations would be integral; in mine, they are only to be added if I can find an artist to create them for me or if I have a surplus of time at the end. The difficulty in creating quality animations would be a massive use of time that could be funnelled into adding more aspects into the game or ensuring that it functions well.
2	Cutscenes	Cutscenes are much the same as above, but even lower priority. Cutscenes would mostly be used to draw interest and to further the story, the latter of which is not integral as the story is a relatively minor part of the game. In addition, cutscenes are even more difficult that animations, especially with pixel-art, due to the detail involved.
3	Cosmetic upgrades	Cosmetic upgrades have been requested by the stakeholders but, for reasons detailed above (page 14) it is very unlikely that I would add it. Once again, cosmetics are the least of my priorities due to the difficulty I would have in creating them, and the fact that they are not integral to the game and many of the users. However, this may change if I find an artist who can work with me.
4	Porting	Porting would allow users to transfer saves over different devices. This is not high priority because it is completely unidentified by the stakeholders as necessary, may be more difficult, and has very little necessity in the game. However, I am considering it a limitation because it might add a nice extra element and allow people to keep their progress if their device is going to be sold, or to save as backups.
5	Sound effects	Sound effects would be used for actions like walking, firing the cannon, enemies, or similar. Adding sound effects would make the game more immersive and the players feel more involved, as well as just making it feel more like a game. However, recording/finding these would take valuable time, as would adding them into the game, and it doesn't impact the game overmuch. Thus, it is unlikely that I will add sound effects, as I would prefer to focus on functionality and depth of gameplay.
6	Music	Music can be key to adding immersion to a game, so a suitable track could really improve the game. However, a single track looping through all three levels would get boring very quickly, and adding it only at tense moments would require a fairly significant time investment in detecting when the user comes in and out of a 'combat state', as well as making any more pacifist players unable to get the same experience. In addition, only having music in some moments can make it feel odd when there is no music, especially if there are no sound effects. In addition to the above, finding the correct track to thematically and aesthetically link to my game would be difficult, and the game itself takes precedence.

HARDWARE & SOFTWARE REQUIREMENTS

Cannonfall is meant to be played on a computer, and I do not plan to add mobile controls. As I am creating this in C# in Unity, the requirements will be to run Unity as a player, as taken from the Unity Manual (Unity, 2025). I will aim to create a Windows-suitable version and perhaps add extra systems if I have time.

Hardware:

Any device with a keyboard will be suitable for the base game.
A basic graphical interface
Speakers may be desirable if game sounds are added.

Software:

Minimum requirements to run the game (Unity, 2025):

Windows:

OS Version: Windows 10 version 21H1 (build 19043) or newer
CPU: X64 architecture with SSE2 instruction set support, Arm64
Graphics API: DX10, DX11, DX12 or Vulkan capable GPUs
Additional: hardware vendor officially supported drivers

Computational Suitability

My solution uses a number of computational methods:

Concurrency will help with processing the movements of the player character, the cannonball projectiles and the enemies simultaneously; the nature of the game as a 2D platformer with pixel-art visuals means that it should be easy to process and therefore run smoothly.

To an extent, **abstraction** will be used. There will be mechanisms that are ignored or simplified for the sake of enjoyment and saving some development time, like not having to reload cannonballs (as loading a cannon takes a lot of time) and the enemies not having working mechanisms, just appearing to be robotic. This means that the user can focus on playing the game and enjoying the challenge, and that I don't have to create a logistically functional way of carrying around and reloading cannonballs, for example. A computational approach means that I can ignore these limitations and increase ease of creation and use. In addition, abstracting some details of the environment would make the visuals cleaner and simpler so that the player can focus on the gameplay easier.

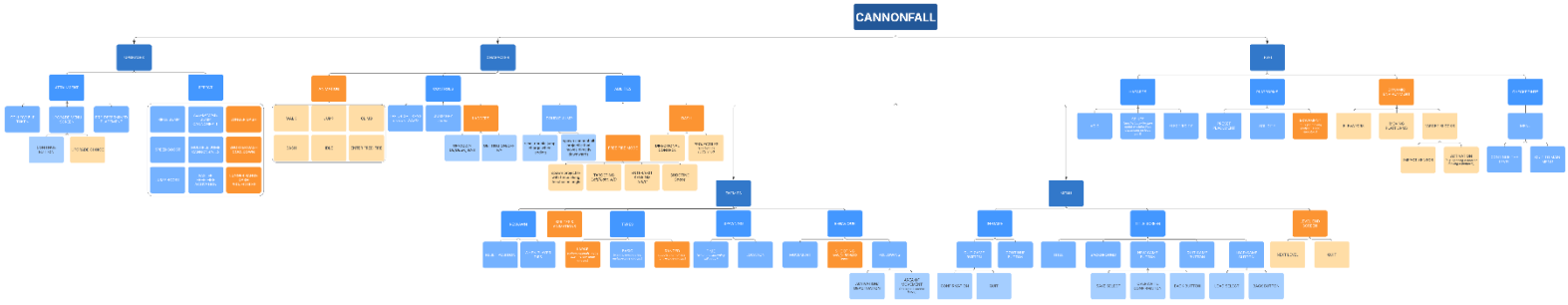
As a game, it is very suitable in regard to **decomposition**; Cannonfall can be split into enemy classes, platforms, the player, etc. Even further, each class can be split into methods, which makes development much easier as I can code and run segments as I go. By approaching the problem computationally, it allows me to simplify development – also, some parts of the game may be used in multiple places, so I can reuse segments to reduce development time and increase program maintainability.

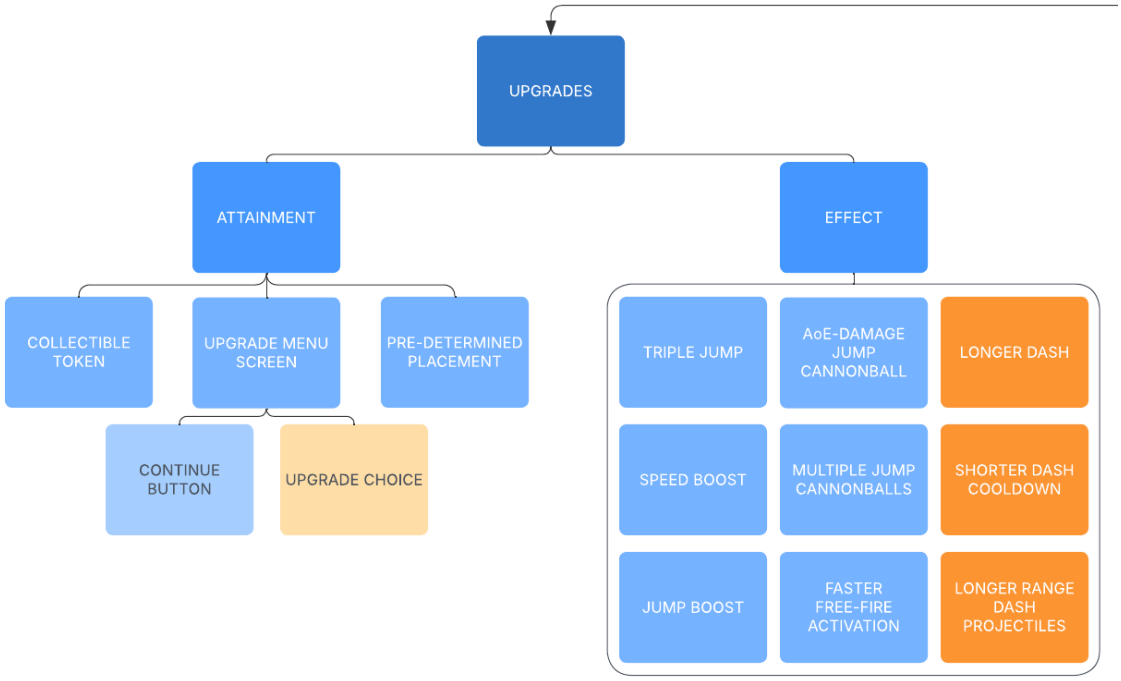
In addition, video games are inherently computational tasks. Without a computer, the user would be forced to use a physical representation (in the vein of a board game). The problems with this are: concurrency, where the user cannot move their character and the enemies simultaneously; mechanics, where actually firing a cannonball could be dangerous and there could be issues with the mechanism and the loading (as avoided by abstraction); the restrictive size of levels; there wouldn't be the same fluidity as given when using controls. These, among other things, encapsulate the reason that Cannonfall is suitable for approach with a computational method.

Design

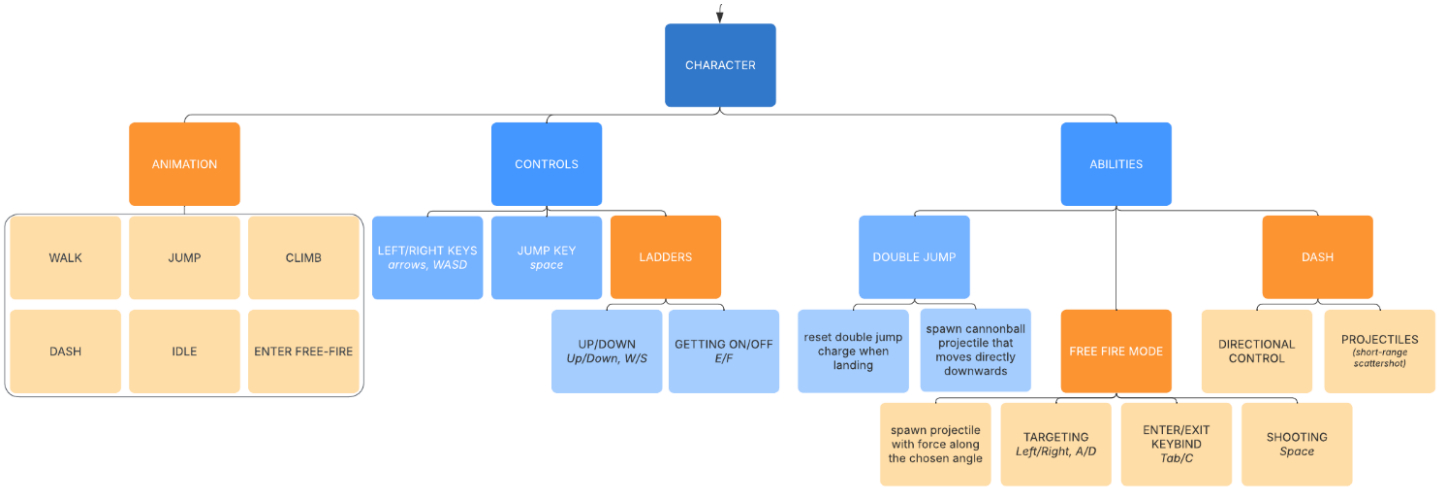
Structure Diagram

OVERVIEW



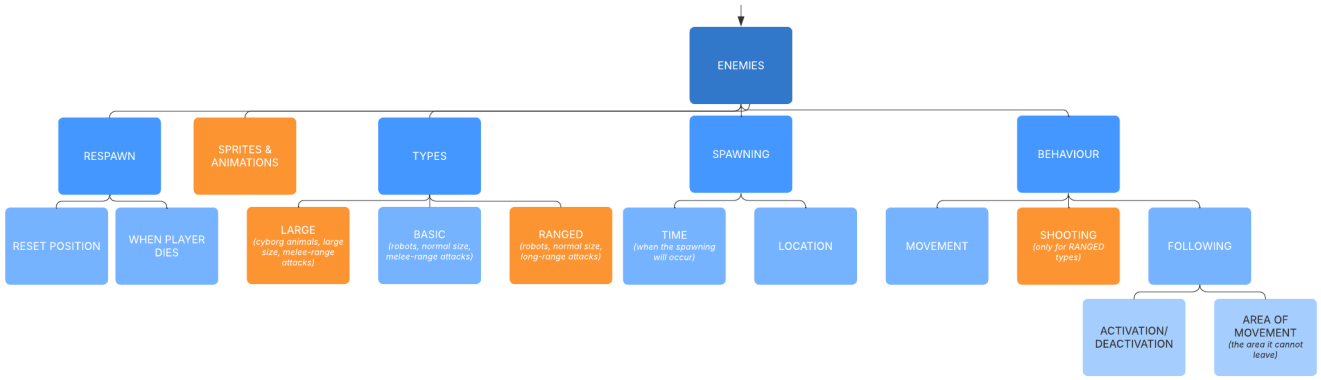


This branch of the structure diagram concerns upgrades. This entire sector will be dealt with over two development stages in order to give myself better deadlines and ensure basic functionality before diving into the more complex ones. The non-blue boxes are desirable, so will be added after these two stages if possible. I decided to isolate this as its own branch because Upgrades are a cohesive part of the game in and of themselves, relying on a new class and methods. However, it also relies on other parts of the game (the things it is modifying) so it will need to be a later stage.



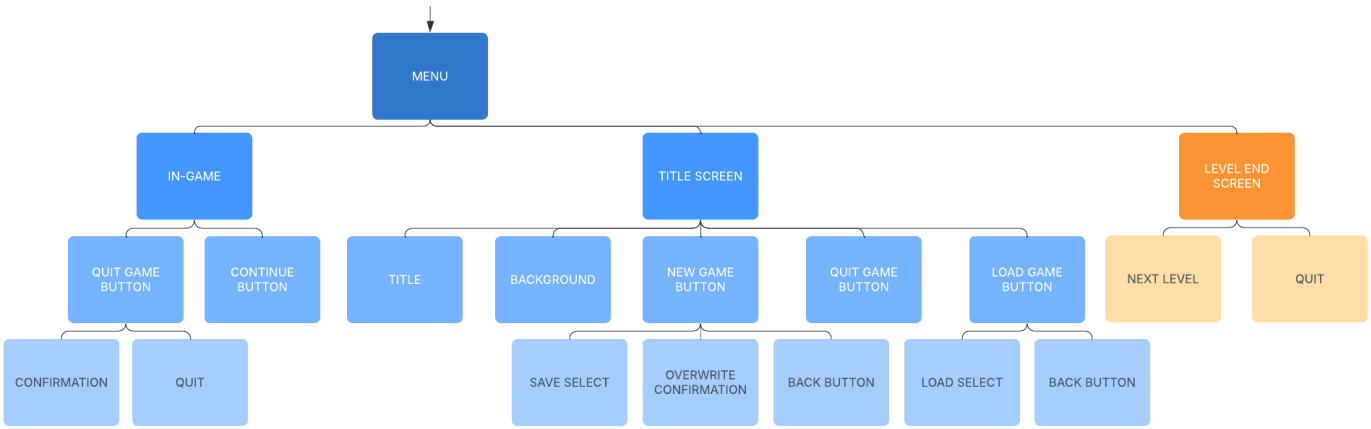
This branch of the structure diagram represents all the functionality that will exist within the player object. I decomposed it in this manner so that I could look at the character at any point and see all the functionality that I have done compared to what needs to be done; i.e. the details above. Again, the orange boxes are lower priority, so may not appear in the game.

I decided to note down possible animations although it is an incredibly low priority so I would think about how to implement them in the design of methods and classes. I also ensured that I had my potential keybinds for abilities and moving, to make it as clear as possible how I was going to use these functions.

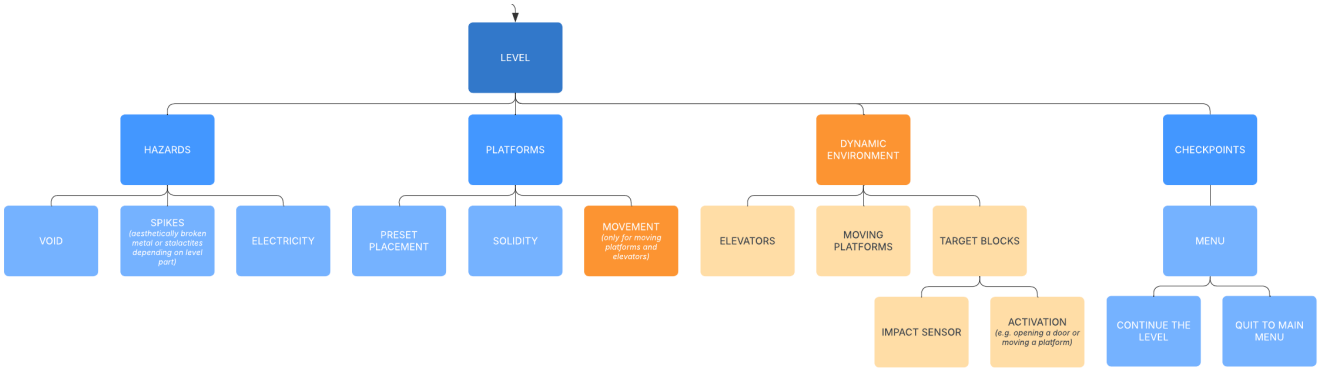


The enemies section is structured in this way because it will be addressed in a single stage of development (if identified as an essential part). This means that the structure diagram holds a list of all the necessary functions of the enemy, including depth on how to spawn or respawn. This allows me to create methods within the enemy class that correspond to the nodes on the structure diagram; the leaf nodes give me greater depth, a checklist of things within that method, or even split the method in two to deal with each part.

The orange boxes have been identified as desirable so may not be added. However, they have been added to the structure diagram anyway due to them inheriting a lot of methods and attributes from the basic class.



This section of the structure diagram details the different menus. They will be explored in greater depth in terms of appearance and justification in the GUI Design section, but the structure diagram has the basic functionality of each screen to ensure that all buttons/selections are taken into account. Each screen is fairly basic, not requiring too much in terms of functionality. The title screen holds nodes which detail two of the screens it can transfer into, in order to show that they are linked. This will also be made clear in the GUI Design section.



The level section contains everything that will be needed before a level can be created. This effectively gates how early I can do a level design stage, because I would like to have the basic components of a level in order to create something I know will function without too many late additions. This is further divided into more specific parts, like the three different hazards. This decomposition keeps me aware of which objects I need to use (e.g. spikes, platform, checkpoint) as well as acting as a checklist I need for a level to take shape.

GUI Design

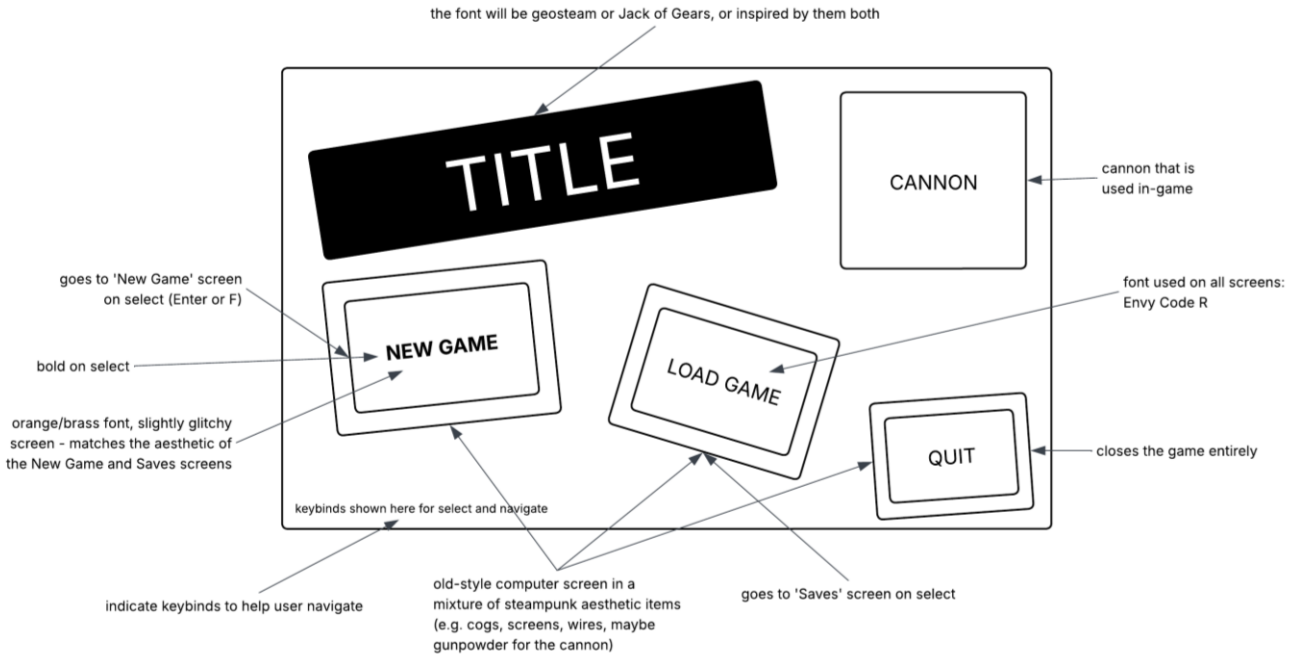
Colour Scheme

The colour scheme will remain consistent throughout the game in order to create a recognisable aesthetic. It should be different enough to be interesting between levels but not diverge so much that it feels like a different game or genre.

Using blacks and greys creates a more dystopian feel to the world, which is common for steampunk. In addition to this, my level environment will be mostly underground; darker colours will lend a sense of claustrophobia and danger. The bronze and orange colours appeal to the often-used brass and cogs which appear in steampunk

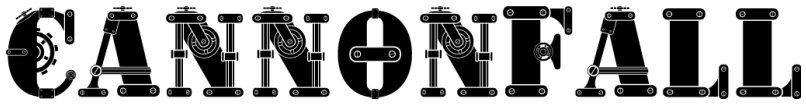
games and media. This creates a strong, relatively simple, and visually appealing aesthetic that is associated with my game. The neon green will be used very sparingly, meant to denote power sources or sections of particular interest, while staying within the overall aesthetic due to being more fluid and less glaringly different than other bright colours (like red and yellow).

Title Screen



Fonts:

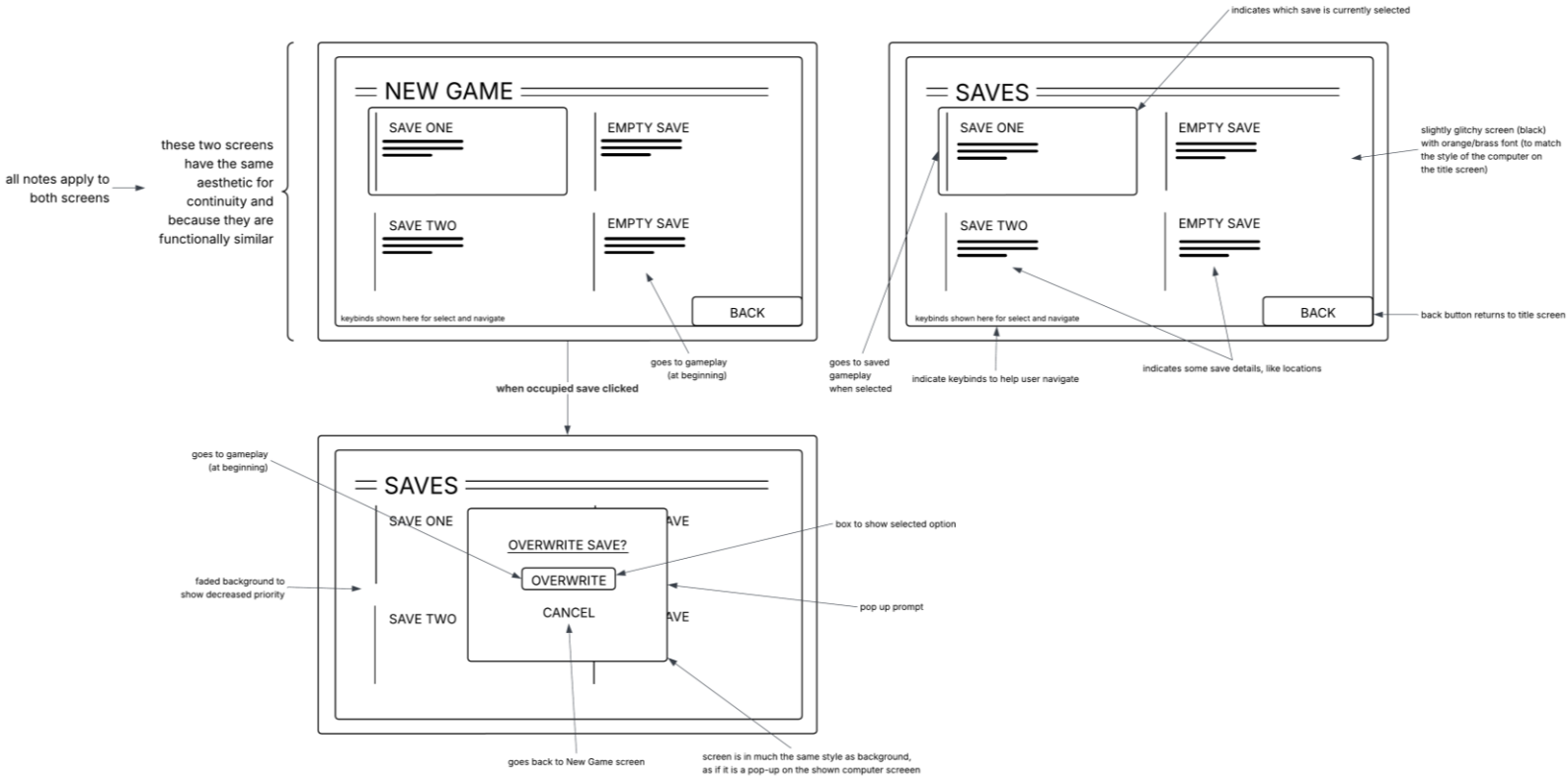
- geosteam:



- Jack of Gears:

The title screen is necessary as an indicator of what the game is and a way to navigate towards playing the game. It will be navigated with the movement keys, as indicated in the bottom left, to keep it accessible for everyone and obvious in how it is used. Generally, it should be a recognisable UI which immediately introduces the user to the game.

Save & Load Menus



Font:

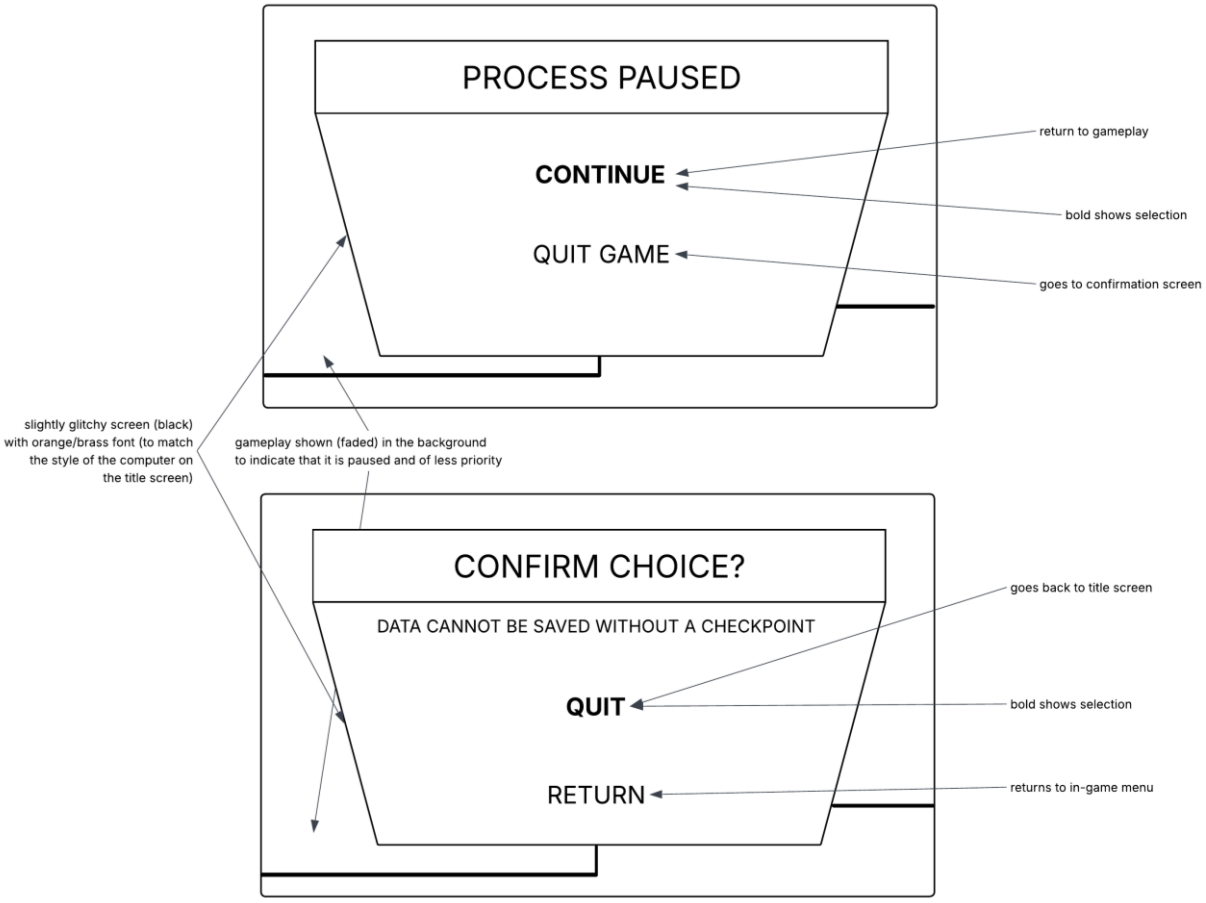
- Envy Code R:

NEW GAME

This font is consistent in all screens as the default, except for the title (as seen above). This serves to increase familiarity and fit with the aesthetic. It also ensures that everything is readable – overly embellished text may be difficult to read, especially for users with dyslexia.

The new game and saves screen is necessary to save user data in a way that is easy to understand. This abstracts the majority of the processes and removes actual interaction with the files, allowing the user to simply pick between four slots. This also allows them to overwrite saves or start new games, in case they don't like their prior playthrough or want to restart. This will be navigated in the same manner as the title screen, to keep consistent and thus increase ease of use.

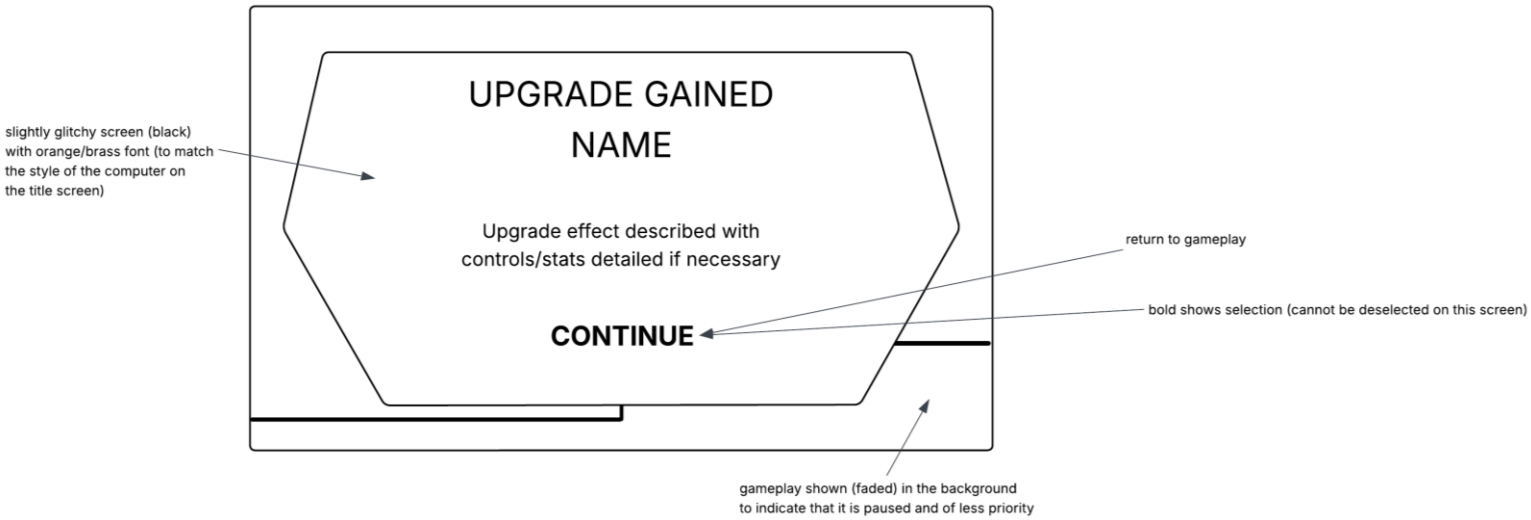
In-Game Pause Menu



The in-game menu is necessary to quit the game, otherwise it would have to be force closed by task manager or the like. Without this key accessibility feature, the game may feel infuriating and would go against its core appeal to my stakeholders – short, challenging sessions.

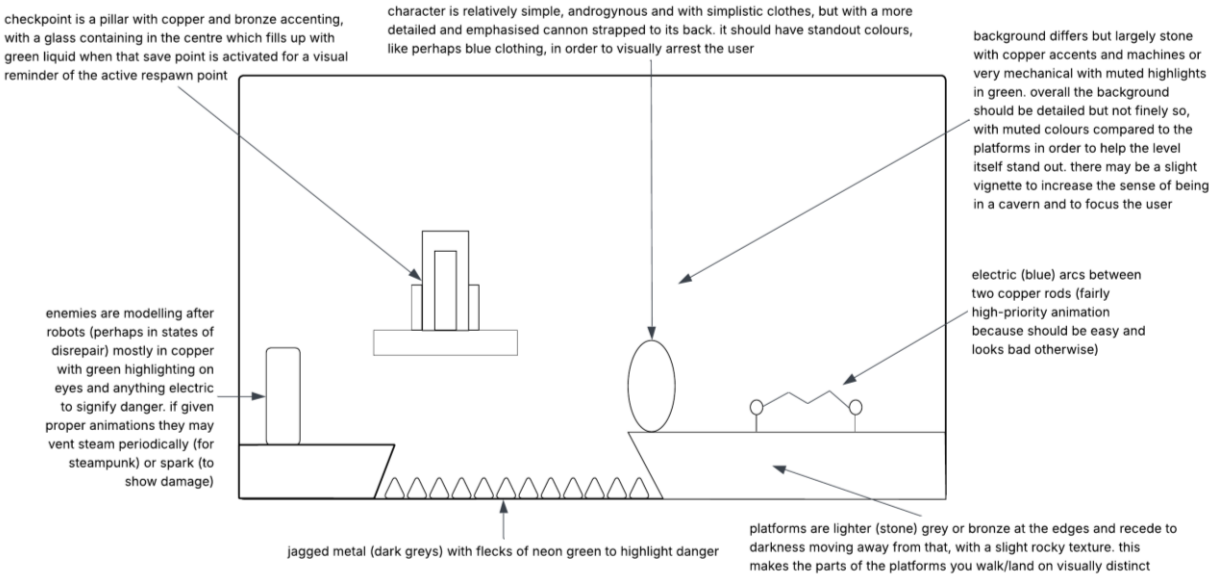
I decided to add a confirmation screen so that users don't accidentally lose progress when they simply wish to pause. This means that retention will be increased, with users more likely to come back and not quit entirely when progress is lost.

Upgrade Screen



This screen is necessary in order to inform the user how the latest upgrade they have gained actually works. This includes keybinds, a description of the upgrade, and perhaps even statistics (if applicable). Without this, it may be tricky to intuitively understand what an upgrade has done, or even notice a difference. It also gives me a good medium to communicate how to use it without a detailed tutorial section/level, which stakeholders said they didn't particularly want.

Gameplay



Development Plan

Stage 1 – Character Movement

Prior to creating any aspect of the game, the character's basic movement must be created and assessed. Without this, the game is entirely non-functional. A basic placeholder for the character's sprite will be displayed on the screen (to be replaced **here**), as well as ensuring that the camera follows the character and that it can double jump. Within this double jump, a cannonball must spawn and be shot downwards

Functionality:

- display a character
- program movement keys

- assign gravity and jump height
- allow for one double jump per jump
- program camera follow

Test No.	Description	Type	Data	Expected Result	Actual Result
1	Character image shown	Valid	Run program	Character sprite is visible in the centre of the screen.	
2a	'Left Arrow' key works	Valid	'Left Arrow' pressed	Character moves left	
2b	'A' key works	Valid	'A' pressed	Character moves left	
2c	'Right Arrow' key works	Valid	'Right Arrow' pressed	Character moves right	
2d	'D' key works	Valid	'D' pressed	Character moves right	
3a	'Sace' key works	Valid	'Up Arrow' pressed	Character 'jumps' upwards from ground	
3b	'Space' key doesn't work if not on ground	Invalid	'Up Arrow' pressed when not on ground	Character does not move	
4	Gravity works	Valid	Run program	Character falls when not on a solid object	
5a	Double jump ('W') works	Valid	'W' pressed when in air	Character jumps again	
5b	Double jump only occurs once	Invalid	Jump key pressed twice in air	Character jumps once	
6	Cannonball sprite appears	Valid	Double jump occurs	Cannonball sprite appears in midair	
7	Cannonball moves	Valid	Double jump occurs	Cannonball moves straight downwards quickly	
8	Cannonball disappears	Valid	Cannonball hits the ground	Cannonball is deleted	

Stage 2 – Hazards and Respawns

This stage is necessary to address a crucial part of the game – respawning. Programming in hazards (the parts of the environment that can kill the character) allows me to see how the character might die and how to transition from there into a respawn. In addition, it lets me start developing the functionality that resets the player to the start. I must also allow the 'void' to kill the player, which also means that the camera needs to have certain bounds so that the player can fall outside them. These will be preliminarily addressed, but will needed to be added in level design (Stage 8 and 12) as the level has varying heights and thus different limits may occur. In addition, the cannonball must be destroyed if it hits a tangible hazard.

Functionality:

- show placeholder sprites for hazards
- give hitboxes to the hazards (not arc)
- 'kill' character on touch
- reset character to beginning
- transition from death to respawn (black screen fade)
- create 'void' hazard
- ensure the cannonball can be destroyed when it hits an object
- code some preliminary camera limits

Test No.	Description	Type	Data	Expected Result	Actual Result
1	'Spikes' sprite shown	Valid	Run program	Spikes are visible on the screen	
2	Spike hitbox	Valid	Character on spikes	Spikes are solid objects that the character can walk on	

3	Spikes can kill	Valid	Character touches spikes	Character sprite is teleported to (**start position**)	
4	Transition functions	Valid	Character touches spikes	Screen fades to black, then fades back to gameplay with character sprite teleported to (**start**)	
5	'Arc' sprite shown	Valid	Run program	Electricity arc is visible on screen	
6	Arc has no hitbox	Invalid	Character inside electricity	Character falls through the arc	
7	Arc can kill	Valid	Character touches arc	Death transition occurs	
8	Void hitbox	Valid	Character on void	The void has a solid object that can be walked on	
9	Void can kill	Valid	Character touches void	Death transition occurs	
10a	Spikes destroy cannonball	Valid	Cannonball touches spikes	Cannonball is destroyed	
10b	Arc does not destroy cannonball	Invalid	Cannonball touches arc	Cannonball goes through arc and is destroyed by ground	
10c	Void destroys cannonball	Valid	Cannonball touches void	Cannonball is destroyed	
11	Camera cannot pan down to see character on void	Valid	Character falls onto void	Camera does not move downwards as character falls into void	

Stage 3 – Checkpoints

This stage heavily relies on the previous stage. Building on the respawn functionality added in the last, checkpoints that serve as places to respawn are added. These must be interacted with in order to set respawn point, where the player will reappear if they touch one of the hazards recently introduced. I need to ensure that respawn points overwrite the position of the last one used, and that they function as intended (only if interacted with). In addition, to show that they need to be interacted with a pop-up showing the key-bind for interacting will appear when the checkpoint is touched.

Functionality:

- checkpoint sprite
- interact pop-up
- sprite change when interacted with
- respawn position changes on interaction

Test No.	Description	Type	Data	Expected Result	Actual Result
1	Checkpoint sprite shown	Valid	Run program	Checkpoint sprite is visible on screen	
2	Checkpoint has no hitbox	Invalid	Run program	Checkpoint cannot be stood on	
3	Interact pop-up appears	Valid	Character touches checkpoint	Pop-up appears over the checkpoint	
4	Alt. Checkpoint sprite shown	Valid	Run program with Active sprite	Alt. Checkpoint sprite is visible on screen	
5a	'F' key works	Valid	'F' pressed while touching checkpoint	Checkpoint switches sprites	
5b	'E' key works	Valid	'E' pressed while touching checkpoint	Checkpoint switches sprites	
6a	Respawn overwrite works	Valid	Character touches hazard after checkpoint interaction	Respawns at interacted checkpoint	

6b	Respawn overwrite works	Valid	Character touches hazard after interacting with two distinct checkpoints	Respawns at second checkpoint	

Stage 4 – Basic Enemy

An enemy is the next step up from a hazard. I chose to program this after the checkpoints were added because enemies need to respawn after the player is killed to maintain challenge level in an area. These enemies must activate when the player comes within a certain distance, follow the player for some distance and be able to traverse the environment. If the above does not occur, they are effectively hazards. To make them a threat, they should be able to do an attack that harms the player at melee range (so that the player can dodge and counter more easily). The player's cannonball needs to be able to destroy the enemies so that the threat can be removed, unlike with hazards, to make sure that the cannonball is an effective weapon.

Functionality:

- display enemy sprite
- create enemy spawn point
- spawn enemies when the player gets close enough
- enemy movement (being able to get up small hills and ledges)
- enemies go towards the player if the environment is traversable
- enemies follow for a certain distance from spawn points
- enemies return to spawn points when player leaves range
- enemies despawn when player gets far enough away
- enemies attack player at close range
- enemies are destroyed by cannonballs (and despawn cannonballs when hit)

Test No.	Description	Type	Data	Expected Result	Actual Result
1					

Stage 5 – Title Screen Buttons

Because the first four stages make up the majority of the necessary gameplay functions, the GUI must now be designed. This stage involves making a mock-up of the GUI and adding in the functionality of the title screen. It should be able to transfer to both of its child screens (i.e. the New Game and Load Game screens) and quit the game entirely. In addition, the user should be able to navigate the screen with the key-binds shown.

Functionality:

- initial GUI design added
- 'button' sprites (with unselected and selected variants)
- menu can be navigated with AD and Left/Right Arrows
- buttons can be selected with F and E
- transfers between requisite screens
- game can be quit

Test No.	Description	Type	Data	Expected Result	Actual Result
1a	GUI shown on screen	Valid	Run program on Title Screen scene	Title Screen GUI visible	
1b	Buttons1 sprite visible	Valid	Run program on Title Screen scene	All text for the 'buttons' visible, with New Game in bold	

2a	‘Right Arrow’ works	Valid	‘Right Arrow’ pressed, Buttons1 sprite	Switches to next buttons sprite (Buttons2)	
2b	‘D’ works	Valid	‘D’ pressed, Buttons2 sprite	Switches to next buttons sprite (Buttons3)	
2c	‘Left Arrow’ works	Valid	‘Left Arrow’ pressed, Buttons3 sprite	Switches to prev. buttons sprite (Buttons2)	
2d	‘A’ works	Valid	‘A’ pressed, Buttons2 sprite	Switches to prev. buttons sprite (Buttons3)	
2e					

Stage 6 – Saving/Loading

To give any true functionality (beyond navigation) to the title screen, the game must be able to load saves or create a new save file. This stage concerns the creation and accessing of these saves, the format in which I will save data, how the saves will be presented on the save screen, and ensuring that checkpoints can save data. I also need to balance the four save slots and make sure that they don’t overwrite each other or cause conflicts. By adding this here, it effectively finalises the title screen (beyond any aesthetic retouches in Stage 12), which ensures that another essential part of the program is complete.

Functionality:

- checkpoints save required game data to the requisite file
- game data includes current checkpoint and any upgrades
- new game select can create a new save file
- new game select can overwrite an existing save file
- load game can read the requisite file to start a game from that position
- save files persist beyond the game being terminated

Test No.	Description	Type	Data	Expected Result	Actual Result
1					

Stage 7 – In-Game Menu

This stage is focused on creating the in-game pause menu, which allows the user to pause the game and quit (without saving). This follows naturally on from creating the GUI of the title screen, as it is a very similar process. I must ensure that there are confirmations for quitting the game so that the user does not accidentally lose progress.

Functionality:

- in-game pause menu GUI
- pause menu can be opened by pressing Esc
- pause menu can be closed by pressing Esc
- pause menu can be closed by selecting continue
- game can be quit to title screen by selecting exit
- confirmation prompt appears when exit selected

Test No.	Description	Type	Data	Expected Result	Actual Result
1					

Stage 8 – Level 1 Design

With this stage complete, the basic structure of the game has become playable. This means that there is one level with enemies and hazards that can be traversed by the user, as well as a number of UIs. This stage is only being done now that the player’s capabilities are implemented and hazards/enemies exist so that I can test the design as it is being created to make sure it is both doable and challenging. It allows me to add things as I go, rather than using placeholders for hazards that I would then need to insert later. In

addition, level 1 should be relatively unaffected by upgrades (maybe one or two at the end) as a way of getting the user accustomed to the controls, so I won't need to rework it when upgrades are added. As mentioned in Stage 2, camera follow limits may need to be hard-coded in this stage to make sure the character falls off screen into the void or that the camera doesn't reveal further parts of the level.

Functionality:

- level 1 is created
- hazards added onto level
- enemies added onto level
- checkpoints added onto level

Test No.	Description	Type	Data	Expected Result	Actual Result
1					

Stage 9 – Upgrades pt. 1

This stage lays out the basic formula for creating an upgrade. It concerns the sprite, the spawning of the collectible token that gives the upgrade, and the fact that token is intangible. In addition to that, an additional UI may need to be created – at this stage it will remain basic as it is not entirely necessary beyond a description of what the upgrade does. I will also add some basic upgrades as proof of concept and as ones to add towards the end of Level 1 and the beginning of Level 2.

Functionality:

- upgrade token sprite added
- upgrade token can be collected
- upgrade screen UI appears
- speed upgrade
- jump height upgrade

Test No.	Description	Type	Data	Expected Result	Actual Result
1					

Stage 10 – Upgrades pt. 2

This stage follows on perfectly from the prior – it takes the framework made in Stage 9 and uses it to make the rest of the possible upgrades at the moment. This being distinct from the previous stage allows me to streamline my tests – the basic formula will have been shown to work, so any issues that arise will be due to the code of the upgrades. It is also necessary that this is done before Level 2 & 3 because their placement will drastically alter the composition of the level, so I must be able to see what the upgrades will do before I design the level to create an appropriately challenging level.

Functionality:

- triple jump upgrade
- area-of-effect damage for cannonball upgrade
- multi-cannonball upgrade

Test No.	Description	Type	Data	Expected Result	Actual Result
1					

Stage 11 – Level 2 and 3 Design

This is effectively the final strictly necessary stage. With this, the base game is entirely completed. All three levels will be fully playable and complete, with checkpoints, enemies, hazards, and upgrades. When doing this stage (especially Level 3), I plan to make notes as to where extra things can be added if

time allows. This will allow me to integrate desirable features more easily at a stage where I should have an intuitive understanding of the game and the capabilities of the character. The camera limits noted in Stage 2 and Stage 8 may also need to be added.

Functionality:

- level 2 created
- level 3 created
- hazards added
- enemies added
- checkpoints added
- upgrade tokens added

Test No.	Description	Type	Data	Expected Result	Actual Result
1					

Stage 12 – Visual Retouch

This stage is a retrospective look on the entire game so far, allowing me to improve GUIs if I feel they are lacking and hopefully replace placeholder sprites with ones that more accurately represent what I wish them to. The reason this is so late is to allow time to create this sprites, as well as it being relatively low priority compared to making sure the game itself functions. This stage also involves adding any animations I deem possible, and is the last of the necessary stages.

Functionality:

- sprites retouched/replaced
- animations added
- GUIs improved

Test No.	Description	Type	Data	Expected Result	Actual Result
1					

Stage 13 – Alpha Testing

I will attempt to play segments of the game and see if any bugs or errors arise. This also allows me to get a feel for the difficulty in a more continuous manner than minor tests in level design stages. Although this will have few formal tests (due to it simply being gameplay) it will serve to highlight issues with the gameplay and ensure a relatively streamlined product goes to the next stage.

Functionality:

- any errors resolved

Test No.	Description	Type	Data	Expected Result	Actual Result
1					

Stage 14 – Beta Testing

To involve my stakeholders and give a new perspective to testing the game, I will disseminate it to a number of stakeholders for them to play through and give feedback on. Any verbal feedback will be noted down, and a form will be included for them to give more detailed answers.

Functionality:

- any errors resolved

Test No.	Description	Type	Data	Expected Result	Actual Result

1					

Desirable Stages

This is just a list of possible things I'll put in if given time (in priority order):

- free-fire mode
- moving platforms
- dash
- ranged enemies
- targets
- upgrades to accommodate new abilities
- level end-screen
- animations

Development

Stage 1 – Character Movement

Prior to creating any aspect of the game, the character's basic movement must be created and assessed. Without this, the game is entirely non-functional. A basic placeholder for the character's sprite will be displayed on the screen (to be replaced at Stage 12), as well as ensuring that the camera follows the character and that it can double jump. Within this double jump, a cannonball must spawn and be shot downwards

Functionality:

- display a character
- program movement keys
- assign gravity and jump height
- allow for one double jump per jump
- program camera follow
- shoot cannonball
- ensure the cannonball can be destroyed when it hits an object

Design

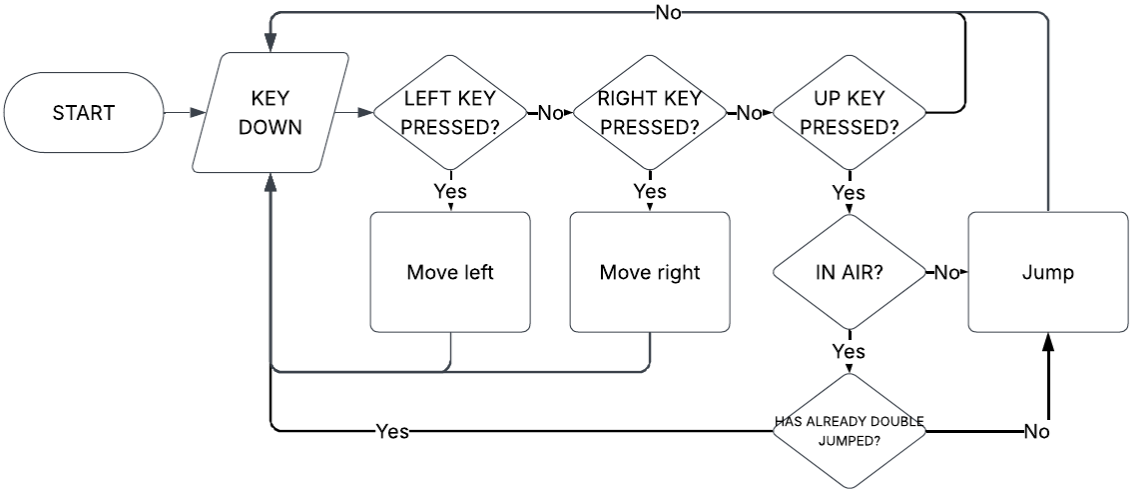
Data Dictionary

NAME	TYPE	Class	Parameters	DESCRIPTION
Update()	Void	Player, CameraFollow, Cannonball		A method which activates every frame (60fps), which allows me to run my other methods constantly
LateUpdate()	Void	CameraFollow		A version of Update that occurs before Update, so there are no glitches in visual quality
Move()	Void	Player		A method containing the code which allows the character to move
Jump()	Void	Player		A method that contains the code which allows the character to jump

OnCollisionEnter2D()	Void	Player	Collision2D collision	A built-in method that is customised to register when a collision occurs, which will set isGrounded to true when the player touches the ground.
jumpForce	float	Player		The force which the player will be propelled upwards with
moveForce	float	Player		The force which the player is moved sideways with – determines speed
isGrounded	Boolean	Player		Boolean which holds whether the play is on the ground or not to see if it can jump
doubleJump	Boolean	Player		Boolean which holds whether the player has double jumped – will be changed in **XXXX**
GROUND_TAG	string	Player		Holds the value of the tag determining the ground, to ensure that no errors are made in typing later in the script
body	Rigidbody2D	Player		Holds the reference to the player's Rigidbody2D
movementX	float	Player		Gets the direction of travel depending on key pressed, from -1 to 1
cannonballObject	GameObject	Player		Holds the reference to the cannonball object from which to spawn other cannonballs
player	Transform	CameraFollow		Holds the reference of the player's transform, which is used to find position
tempPos	Vector3	CameraFollow		A temporary variable which will be used more in the future to apply limits

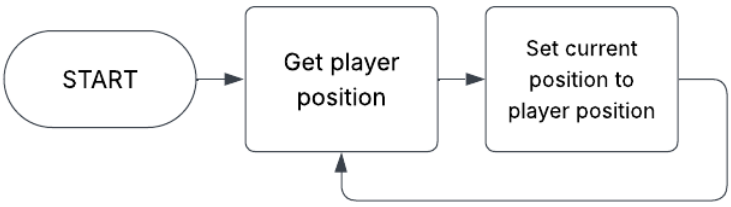
Flowcharts

Movement



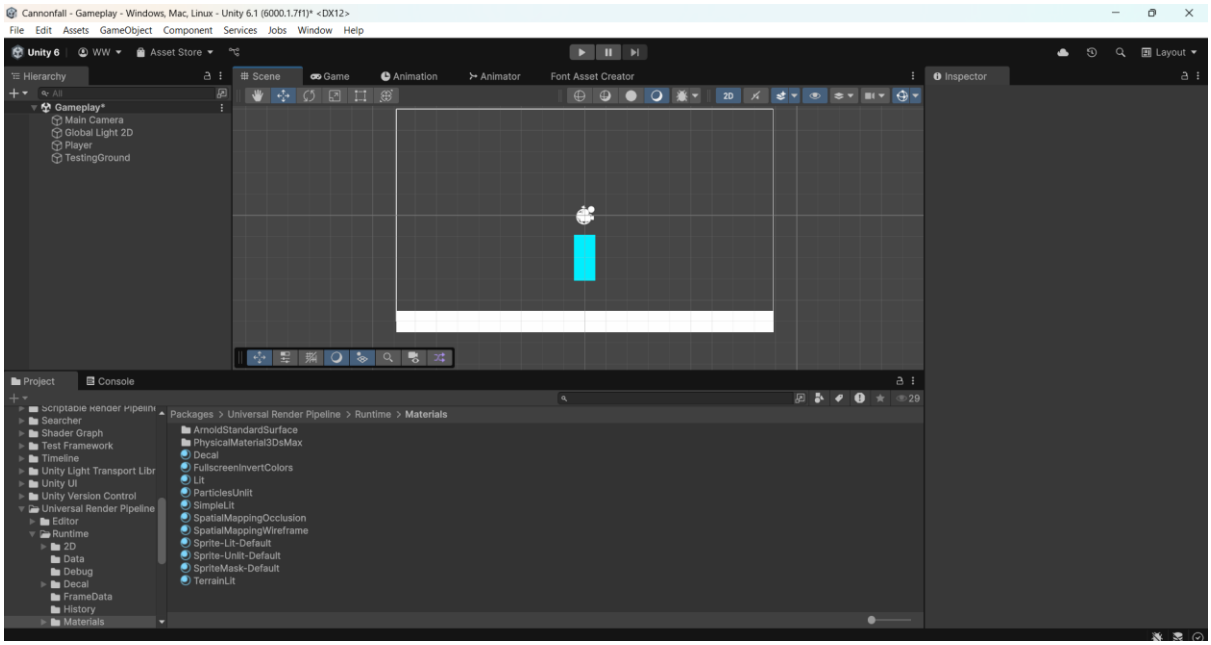
This flowchart represents the process of the character's movement. This can happen concurrently, if more than one key is pressed down simultaneously – this process occurs for each key pressed down.

Camera Follow



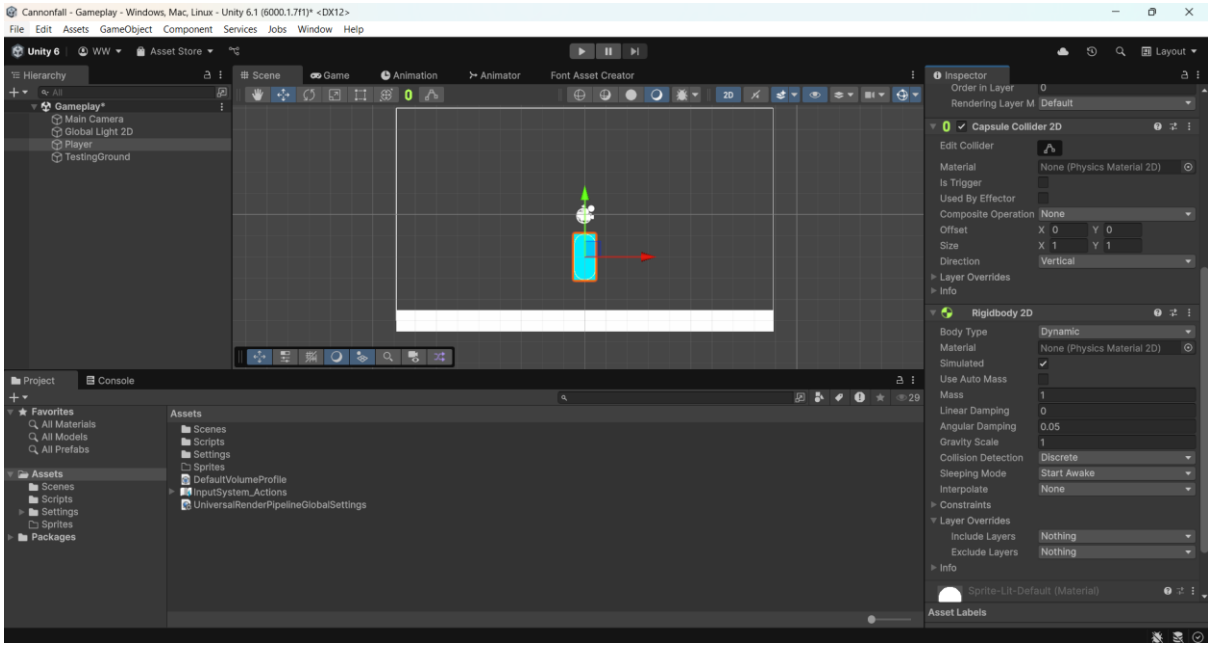
This procedure executes in the LateUpdate() function, so that the camera position is calculated before anything else and there is no jittering. This repeats consistently. I expect the Camera Follow function to be updated in the future to add limits.

Development

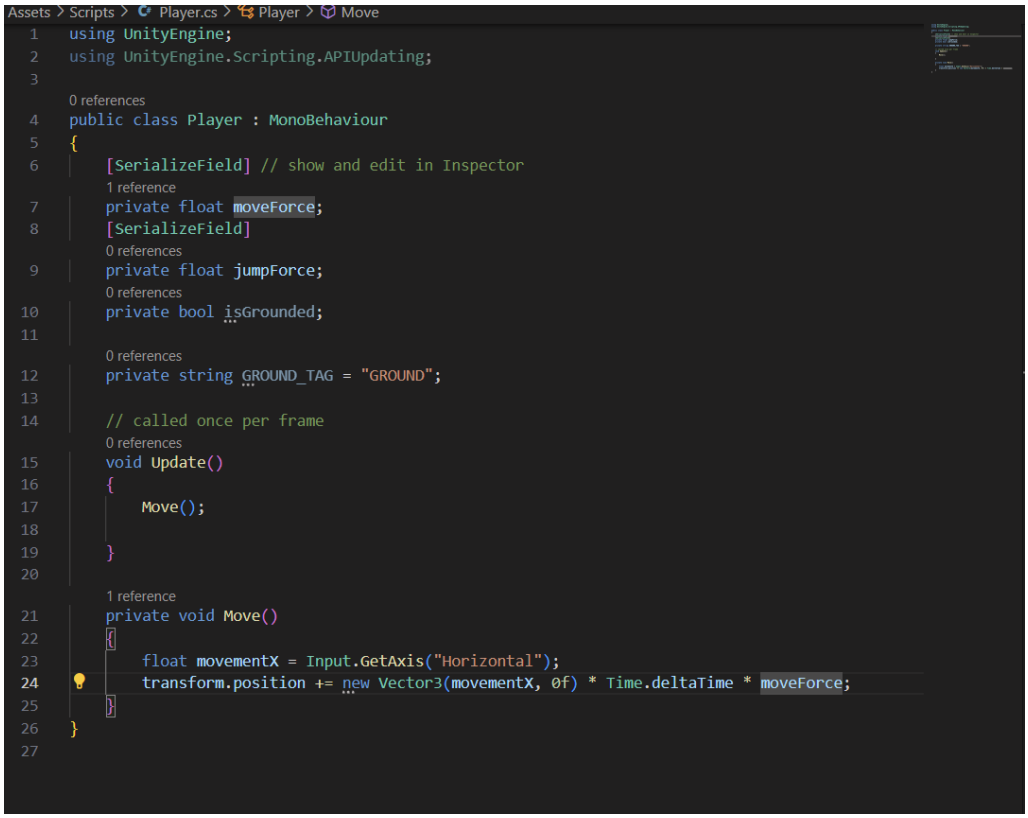


The first thing I did was add a Player object with a placeholder sprite (to fulfil the first functionality objective) and add a TestingGround object. The latter will have nothing but a BoxCollider2D and be tagged

as ‘GROUND’, so I can test more effectively (without the player falling off the screen).

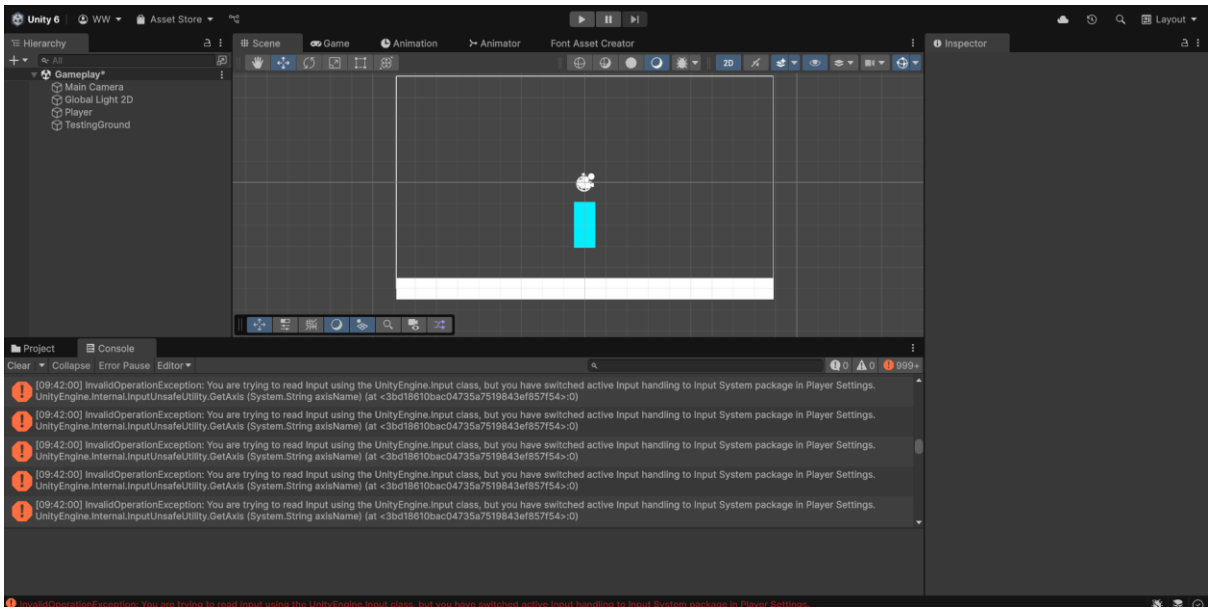


I added a RigidBody2D to the object in order to simulate gravity on my player. I also added a CapsuleCollider2D – the reason it is a capsule and not a box is because I feel as if a capsule will fit the approximate shape of the actual sprite (when it is added) better than a box, although it is currently not representative. This means that I won’t have to remember to change it later, which might also impact other parts of the code if they reference the player’s collider. In addition, capsule colliders don’t get caught on the terrain as easily as box colliders because they are rounded, which makes them a better fit in my platformer, where the terrain will likely differ dramatically between sections.



I added the majority of the Player attributes shown in my data dictionary, as well as adding the move function in Update and creating the function. I set both moveForce and jumpForce to be 5 in the Inspector

as a placeholder value.



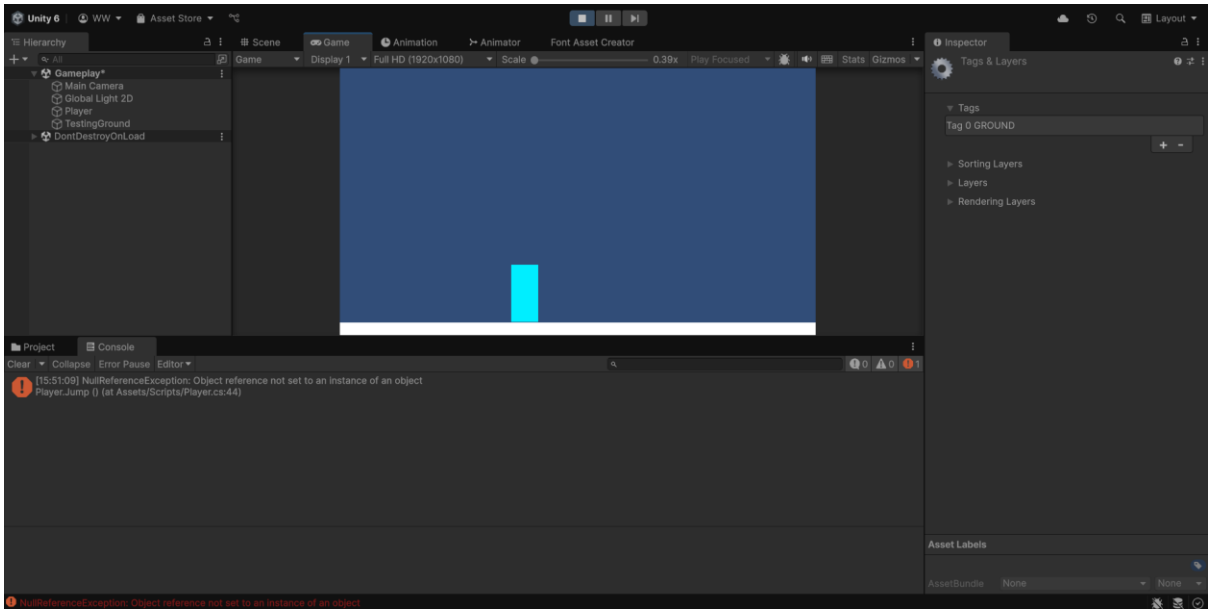
Upon running, I realised that I was using the old version of Unity input handling, so I had to change it in the player settings. I'm using the old version because I find it more intuitive – the tutorials I encountered while learning Unity tended towards this version, and it fulfils all the necessary functions.

When run again, the player could move. However, it was glacially slow – after playing around a little, I settled on 11 as a provisional value.

```
private void Jump()
{
    if (Input.GetButtonDown("Jump") && isGrounded) // if jump button and on ground
    {
        isGrounded = false;
        body.AddForce(new Vector2(0, jumpForce), ForceMode2D.Impulse);
    }
    if (Input.GetButtonDown("Jump") && doubleJump && !isGrounded) // if not on ground but has a double jump charge
    {
        doubleJump = false;
        body.AddForce(new Vector2(0, jumpForce), ForceMode2D.Impulse);
    }
}

0 references
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag(GROUND_TAG))
        isGrounded = true; doubleJump = true;
}
```

The next step was to add the Jump function. I decided to use an impulse to better represent how real life jumps work. It applies an instantaneous force, which launches the player upwards; the jump would feel less natural if the force was applied over time. I also added the built-in collider function, which is used to check when collisions occur. This means that when the player touches the ground it regenerates both Booleans.

I decided to use the doubleJump Boolean to check if the player had double jumped to ensure that it could only happen once per jump. I plan to change this to an integer value to interact with the triple jump upgrade when that stage occurs, but it would simply be over-complicating at the moment.



This error occurred because I didn't reference the Rigidbody2D properly. I added the Awake() function, which happens when the player is created. Within this method, I used the GetComponent<> function to obtain the reference to the Rigidbody2D, which I applied to my attribute.

```
0 references
void Awake()
{
    body = GetComponent<Rigidbody2D>(); // get the reference to the Rigidbody2D
}
```

```
1 reference
private void Jump()
{
    if (Input.GetButtonDown("Jump") && doubleJump && !isGrounded) // if not on ground but has a double jump charge
    {
        body.linearVelocity = new Vector3(body.linearVelocityX, 0f, 0f); // continues with x velocity, resets y to 0
        doubleJump = false;
        body.AddForce(new Vector2(0, jumpForce), ForceMode2D.Impulse);
    }
    if (Input.GetButtonDown("Jump") && isGrounded) // if jump button and on ground
    {
        isGrounded = false;
        body.AddForce(new Vector2(0, jumpForce), ForceMode2D.Impulse);
    }

}
```

After running again, the double jump was not working properly. I switched the execution of the scripts around, because they were both activating simultaneously, as well as remembering to add a line which cancelled the vertical velocity. Without this, the second jump was happening at effectively the same time as the first, and the upwards force of the second jump would have had to combat the downwards velocity from falling (therefore jumping less than the first jump).

```
0 references
public class CameraFollow : MonoBehaviour
{
    2 references
    private Transform player;
    2 references
    private Vector3 tempPos;

    0 references
    void Awake()
    {
        player = GameObject.FindWithTag("PLAYER").transform; // gets reference for player transform

    }
    0 references
    void LateUpdate() // executes before update every frame
    {
        tempPos = player.position;
        transform.position = tempPos;
    }
}
```

I added the provisional camera follow code next. The code in Awake() gets a reference from the player for its position, which the camera's position is set equal to. I decided to do the inefficient two part method as seen in LateUpdate() because it will be simpler to update in later stages, when camera limits are added (which will likely need to be hard-coded). I also edited the ground tag to be "Ground" instead of "GROUND", to better fit with existing Unity formatting.

```
0 references
void LateUpdate() // executes before update every frame
{
    tempPos = player.position;
    if (tempPos.y < 0)
        tempPos.y = 0;
    tempPos.z = -10;
    transform.position = tempPos;
}
```

When run, the camera did not show anything. I added a line of code ensuring that the z-coordinate remains at -10, where the camera needs to be in order to render the gameplay. The limit in the if statement shown above ensures that the camera doesn't dip below the level of the testing ground – a good example of the sorts of limits that will be added later.

```
using UnityEngine;

0 references
public class Cannonball : MonoBehaviour
{
    0 references
    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("GROUND"))
            Destroy(gameObject);

    }
}
```

```
if (Input.GetButtonDown("Jump") && doubleJump && !isGrounded) // if not on ground but has a double jump charge
{
    body.linearVelocity = new Vector3(body.linearVelocityX, 0f, 0f); // continues with x velocity, resets y to 0
    doubleJump = false;
    body.AddForce(new Vector2(0, jumpForce), ForceMode2D.Impulse);
    GameObject cannonball = Instantiate(cannonballObject); // create a cannonball
    cannonball.transform.position = body.transform.position; // sets it position to that of the player
}
```

I added the cannonball object, and wrote a basic script. This should make an intangible object that travels downwards (due to assigning it a rigidbody) and destroys itself on contact with the ground. I also added an instantiate line to the double jump portion, which should spawn a cannonball by finding the object with that reference. It then teleports the cannonball to the position of the player to create the appearance that the cannonball is fired from the player.

Testing

Test No.	Description	Type	Data	Expected Result	Actual Result
1	Character image shown	Valid	Run program	Character sprite is visible in the centre of the screen.	Stage 1.mp4 00:00
2a	'Left Arrow' key works	Valid	'Left Arrow' pressed	Character moves left	Stage 1.mp4 00:00
2b	'A' key works	Valid	'A' pressed	Character moves left	Stage 1.mp4 00:01
2c	'Right Arrow' key works	Valid	'Right Arrow' pressed	Character moves right	Stage 1.mp4 00:03
2d	'D' key works	Valid	'D' pressed	Character moves right	Stage 1.mp4 00:04
3	'Space' key works	Valid	'Up Arrow' pressed	Character 'jumps' upwards from ground	Stage 1.mp4 00:05
4	Gravity works	Valid	Run program	Character falls when not on a solid object	Stage 1.mp4 00:05
5a	Double jump works	Valid	'W' pressed when in air	Character jumps again	Stage 1.mp4 00:06

5b	Double jump only occurs once	Invalid	Jump key pressed twice in air	Character jumps once	Stage 1.mp4 00:09
6	Cannonball sprite appears	Valid	Double jump occurs	Cannonball sprite appears in midair	Stage 1.mp4 00:06
7	Cannonball moves	Valid	Double jump occurs	Cannonball moves straight downwards quickly	Stage 1.mp4 00:06
8	Cannonball disappears	Valid	Cannonball hits the ground	Cannonball is deleted	Stage 1.mp4 00:07

Evaluation

In this stage, I wanted to:

- display a character
- program movement keys
- assign gravity and jump height
- allow for one double jump per jump
- program camera follow
- shoot cannonball
- ensure the cannonball can be destroyed when it hits an object

I believe that all of these points were fulfilled completely, to an extent that all these would function in the game. I will redo the sprite in Stage 12 and may tinker with jump height and restrictions on camera follow throughout, but their base code is acceptable for the basic version of the game. I'm happy with the code I created, and believe it is tested such that it will not raise errors later.

Stage 2 – Hazards and Respawns

This stage is necessary to address a crucial part of the game – respawning. Programming in hazards (the parts of the environment that can kill the character) allows me to see how the character might die and how to transition from there into a respawn. In addition, it lets me start developing the functionality that resets the player to the start. I must also allow the 'void' to kill the player, which also means that the camera needs to have certain bounds so that the player can fall outside them. These will be preliminarily addressed, but will needed to be added in level design (Stage 8 and 12) as the level has varying heights and thus different limits may occur. In addition, the cannonball must be destroyed if it hits a tangible hazard.

Functionality:

- show placeholder sprites for hazards
- give hitboxes to the hazards (not arc)
- 'kill' character on touch
- reset character to beginning
- transition from death to respawn (black screen fade)
- create 'void' hazard
- ensure cannonball is destroyed on contact with hazards
- code some preliminary camera limits

Design

Data Dictionary

NAME	TYPE	Class	Parameters	DESCRIPTION
Update()	Void	Spikes, Arc, Void		A method which activates every frame (60fps), which allows me to run my other methods constantly

zapCycle()	Void	Arc		Controls the turning on and off of the electricity beam using a given time
zapCycleTime	float	Arc		Time value which one cycle takes (from turning on to turning off, or from turning off to turning on)
arcOn	bool	Arc		Records whether the arc is on or off
arcCollider	BoxCollider2D	Arc		Holds the reference to the collider
animator	Animator	Arc		Holds the reference to the animator
playerDeath()	Void	Player		Handles the player dying and the respawn process
HAZARD_TAG	string	Player		Holds the tag for the hazards, in case I change it later and to keep it consistent

Pseudocode

```
CLASS Player
    VECTOR3 respawnPoint = (0,0,0)

    PROCEDURE OnCollide()
        IF collidesWith.CompareTag(HAZARD_TAG) THEN
            playerDeath()
        ENDIF
    ENDPROCEDURE

    PROCEDURE playerDeath()
        INSTANTIATE(fadeToBlack)
        transform.position = respawnPoint
    ENDPROCEDURE
ENDCLASS
```

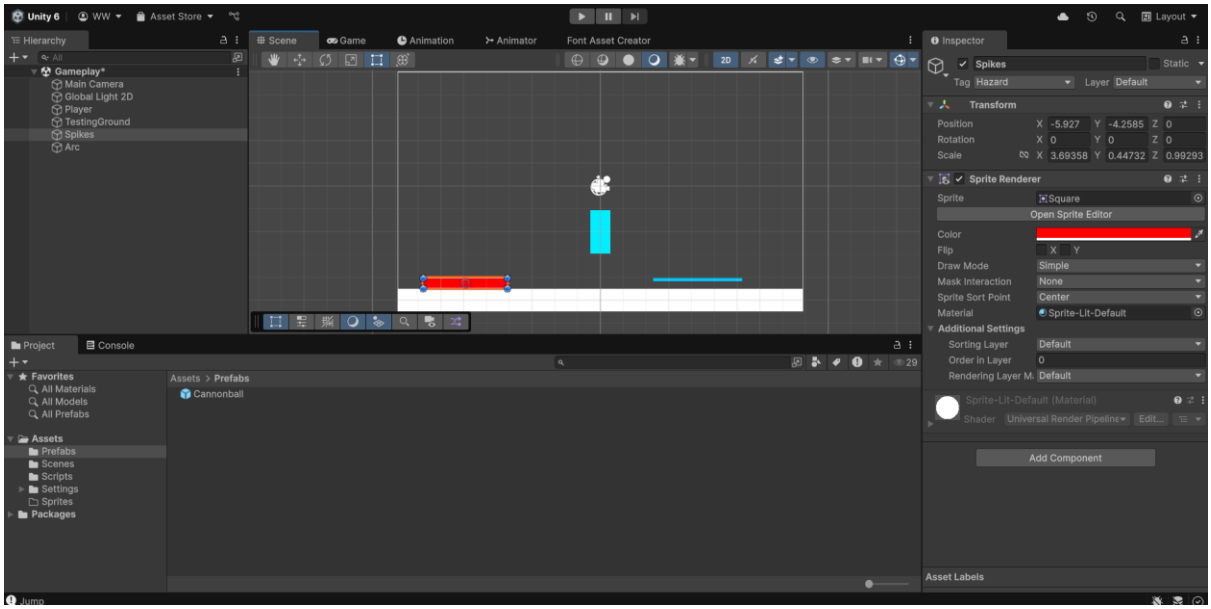
^added to Player script

```
CLASS Cannonball
    PROCEDURE OnCollide()
        IF collidesWith.CompareTag("Hazard") THEN
            DESTROY(self)
        ENDIF
    ENDPROCEDURE
ENDCLASS
```

^added to Cannonball script

```
CLASS Arc
    PROCEDURE ZapCycle()
        WAIT zapCycleTime
        IF arcOn THEN
            arcCollider.isEnabled = false
        ELSE
            arcCollider.isEnabled = true
        ENDIF
    ENDPROCEDURE
ENDCLASS
```

Development



First thing was to add the placeholder sprites for both visible hazards. I also tagged them with the 'Hazard' tag, which will be used later. This tag is constant for all three hazards in order to simplify the response from the character – it will 'die' no matter what, and I do not plan to add animations, especially not detailed ones (like different ones for each hazard).

```
public class Arc : MonoBehaviour
{
    [SerializeField]
    1 reference
    float zapCycleTime;
    1 reference
    bool arcOn = true;
    1 reference
    private BoxCollider2D arcCollider;


    0 references
    void Awake()
    {
        arcCollider = GetComponent<BoxCollider2D>();
    }
    0 references
    void Update()
    {
        StartCoroutine(zapCycle());
    }

    1 reference
    IEnumerator zapCycle()
    {
        while (true)
        {
            yield return new WaitForSeconds(zapCycleTime); // wait for this much time

            if (arcOn) // if electric is on
            {
            }
            else
            {
            }
        }
    }
```

I added all the attributes that are used in the pseudocode, ensuring that the arc begins in the 'on' position. When the arc is created, it gets the reference for its own collider. In update, I start the Coroutine that determines the turning on and off of the arc. I made it an IEnumerator because it is a simple way to create a time delay, which is necessary for a delay between turning on and off. The skeleton of the method to turn it on and off has also been created. I plan to add an animator later just to switch between the 'on' sprite and 'off' sprite, so will add the code framework necessary to do so.

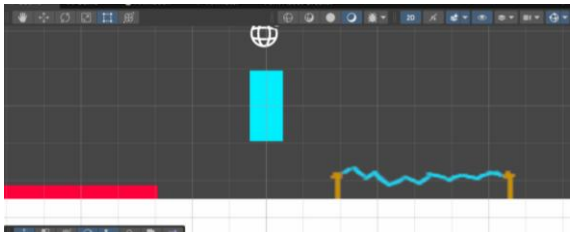
At the moment, the arcs will all be loaded and begin to undergo their processes as soon as the level is. This is because I don't believe my game will be overly resource-intensive, so it should be fine. If it proves to be lagging the game, this will be changed in a later stage.

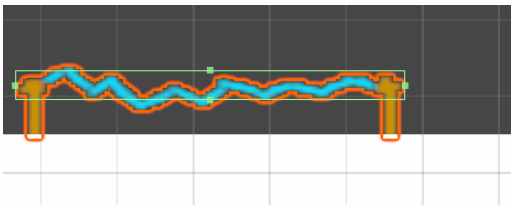
```
private Animator animator;


0 references
void Awake()
{
    arcCollider = GetComponent<BoxCollider2D>();
    animator = GetComponent<Animator>();

}
0 references
void Update()
{
    StartCoroutine(zapCycle());
}

1 reference
IEnumerator zapCycle()
{
    while (true)
    {
        yield return new WaitForSeconds(zapCycleTime); // wait for this much time
        arcOn = !arcOn; // swap the current state of the arc
        if (arcOn) // if electric is on
        {
            animator.SetBool("Arc", true); // set parameter in animator to true
        }
        else
        {
            animator.SetBool("Arc", false); // set parameter in animator to false
        }
    }
}
```

Added in the framework for interacting with the animator. When the Arc parameter is True it will show the 'On' sprite, and vice versa. In addition, I ensured that the state would flip every time interval. This had to be before the if statements to ensure that they updated to the correct state – if it was after, it would be the opposite.

```
if (arcOn) // if electric is on
{
    animator.SetBool("Arc", true); // set parameter in animator to true
    arcCollider.enabled = true; // turns collider on
}
else
{
    animator.SetBool("Arc", false); // set parameter in animator to false
    arcCollider.enabled = false; // turns collider off
}
```

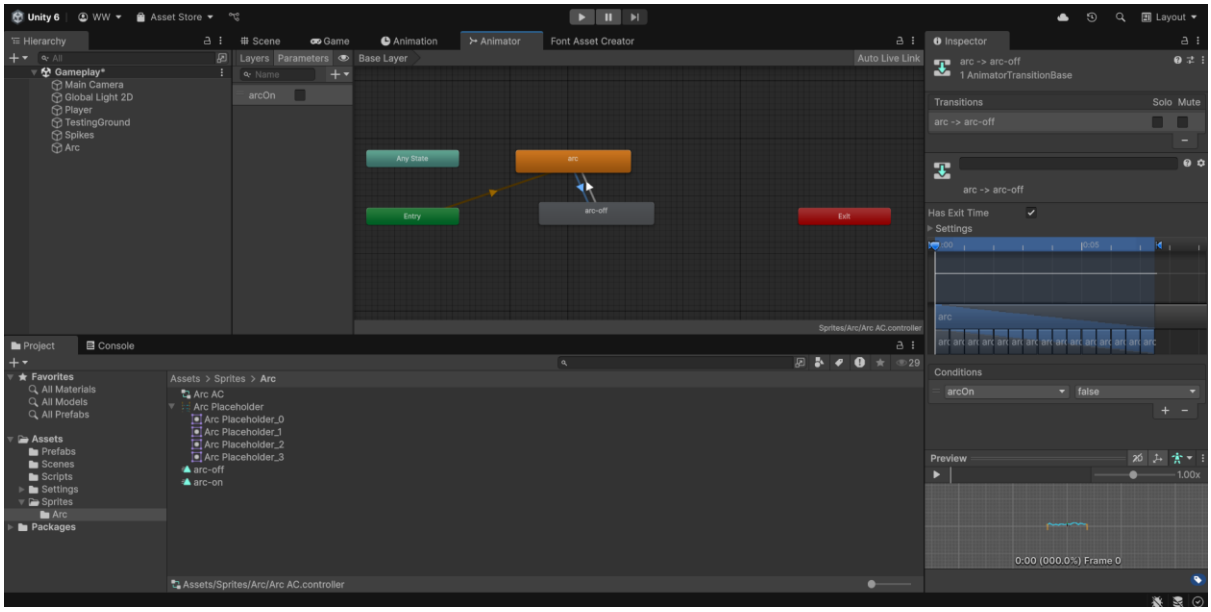
The next step was to ensure that the collider can be turned on and off. This is because the arc will be intangible either way (using the IsTrigger attribute), but while the collider is on the player will act as if it has hit the arc even it is not visibly there. My solution to this is to simply disable the collider while the arc is off, so there is no way it can collide.



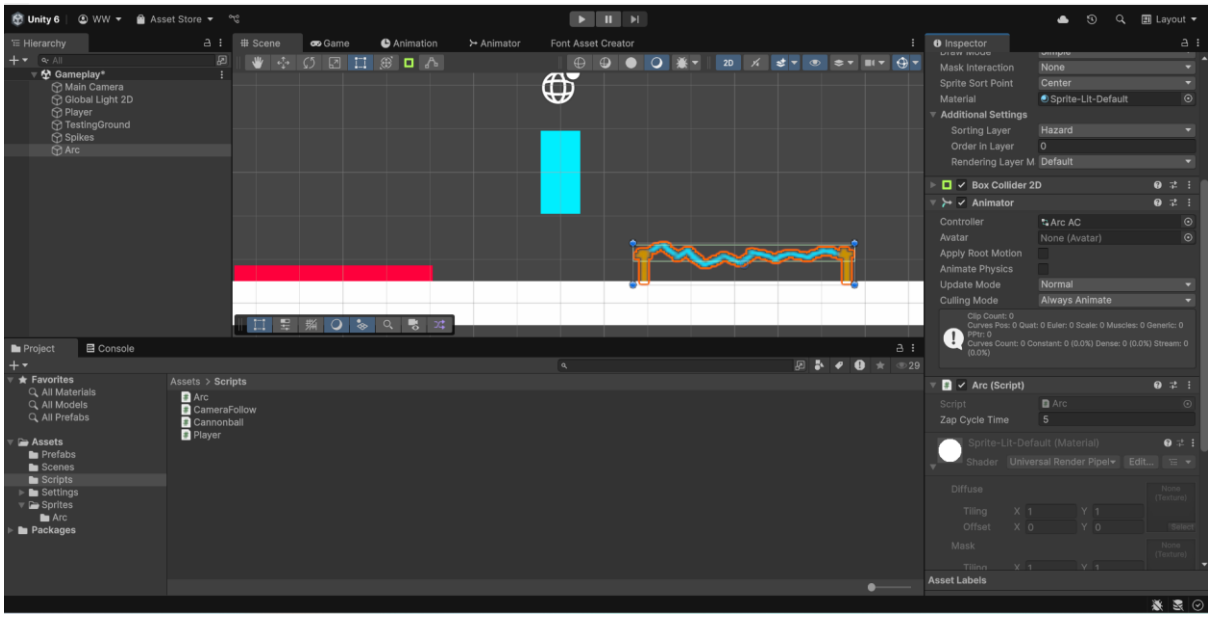


To begin updating the sprite and testing the rudimentary on/off animation, I added a placeholder sprite. This has 4 different sprites – 'off', and then three stages of 'on' which should be meshed together to create a more 'electric' look. I also added the IsTrigger collider to the electric area of the arc. This is kept tight to the line between the nodes to make it more consistent – because the arc will move, the area of danger will remain constant so the player does not have to watch out for every little lick of electricity.

My first attempt at animating the 'on' state was far too quick and looked unnatural. Upon changing the animation speed to 0.11 and the sample frames to 60 (determined by just playing around a little) the placeholder for the arc 'on' animation was determined suitable.



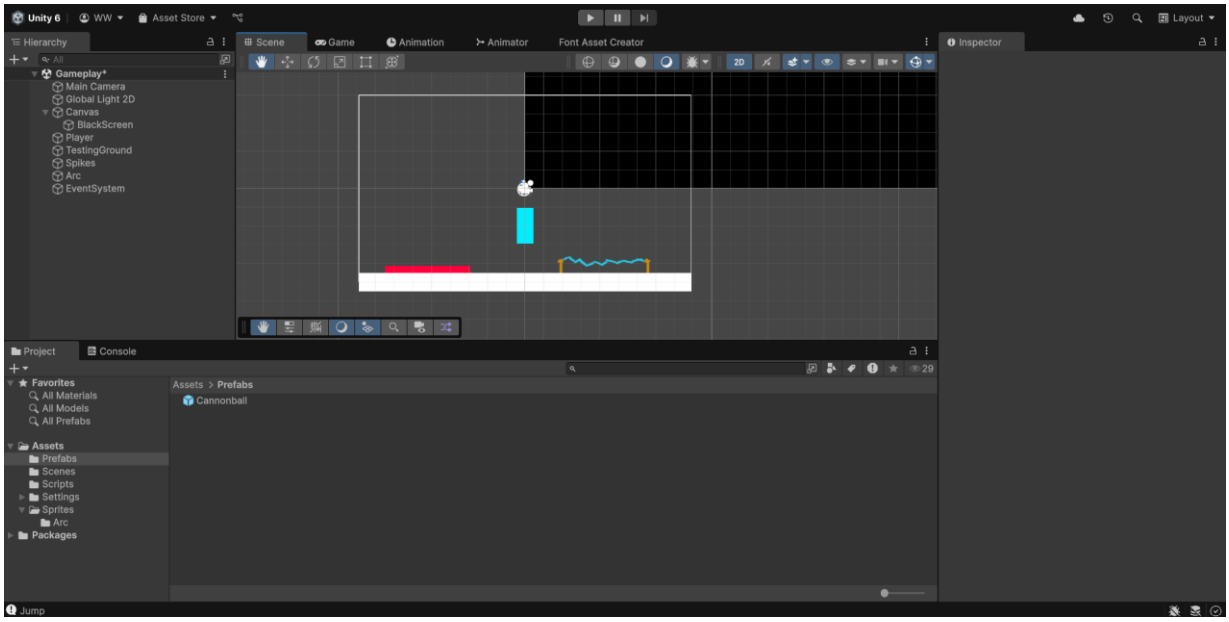
After adding the single-sprite ‘animation’ for the ‘off’ state and adding a transition, I set it to be controlled by the arcOn variable that is used in the Arc class. This should transfer from arc-on to arc-off when it is set to false and vice versa.



I added the script to the Arc object and set the time to 2.5 seconds as a placeholder value just to check. When run, the code appeared to get stuck on the ‘off’ state. The arc disappeared but the collider was rapidly activating and deactivating. Using the console to debug, I found that I had mis-typed the parameter name. However, the delay also seemed to not be working. This was because I had called the start of the Coroutine in Update, so it was being called every frame. Thus, the delay didn’t actually matter because it was stacking up one execution per frame anyway. I moved it to the Start function, where the loop inside the Coroutine would be the determiner, not the fps.

```
0 references
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag(GROUND_TAG))
    {
        isGrounded = true;
        doubleJump = true;
    }
    if (collision.gameObject.CompareTag(HAZARD_TAG))
    {
        Debug.Log("player has died");
    }
}

0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag(HAZARD_TAG))
    {
        Debug.Log("player has died");
    }
}
```

The logical next step was to ensure that the player could recognise when it was touching a hazard. I decided to make the arc intangible because it doesn't make sense for electricity to collide in the same way as spikes or the ground. This will make it look more realistic in the moments before the death transition.



I followed the following tutorial to create the fade-to-black death screen:
https://discover.hubpages.com/technology/How-to-Fade-to-Black-in-Unity
I created a UI canvas and added a black screen with a Screen Space – Overlay, in order to cover the whole screen.

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

0 references
public class FadeToBlack : MonoBehaviour
{
    [SerializeField]
    0 references
    private GameObject blackScreen; // reference to object
    [SerializeField]
    0 references
    private float fadeSpeed;

    0 references
    public IEnumerator FadeInBlackScreen(bool fadeToBlack)
    {
        yield return new WaitForEndOfFrame();
    }
}
```

I followed the tutorial from there, using variable names that made more sense to me and serialising the fadeSpeed so I can edit it from the Inspector. Instead of using a public variable as shown in the tutorial, I serialised the GameObject that will hold the blackScreen; this is for security's sake, although it doesn't have much impact. The standard is generally to use private variables unless necessary.

```
0 references
public IEnumerator FadeInBlackScreen(bool fadeToBlack)
{
    Color objectColor = blackScreen.GetComponent<Image>().color; // get reference to colour
    float fadeAmount;
    if (fadeToBlack) // if fading to black
    {
        while (blackScreen.GetComponent<Image>().color.a < 1) // continue until fully opaque
        {
            fadeAmount = objectColor.a + (fadeSpeed * Time.deltaTime); // how much the thing should fade by
            objectColor = new Color(objectColor.r, objectColor.g, objectColor.b, fadeAmount); // change colour by fade amount
            blackScreen.GetComponent<Image>().color = objectColor; // set this colour to the screen
            yield return null; // stop code when completed loop
        }
    }
}
```

The fade-in section is now complete, with comments for maintainability and Americanised spelling to conform to Unity's standard.

```
0 references
public IEnumerator FadeBlackScreen(bool fadeToBlack)
{
    Color objectColor = blackScreen.GetComponent<Image>().color; // get reference to colour
    float fadeAmount;
    if (fadeToBlack) // if fading to black
    {
        while (blackScreen.GetComponent<Image>().color.a < 1) // continue until fully opaque
        {
            fadeAmount = objectColor.a + (fadeSpeed * Time.deltaTime); // how much the thing should fade by
            objectColor = new Color(objectColor.r, objectColor.g, objectColor.b, fadeAmount); // change colour by fade amount
            blackScreen.GetComponent<Image>().color = objectColor; // set this colour to the screen
            yield return null; // stop code when completed loop
        }
    }
    else // if fading from black
    {
        while (blackScreen.GetComponent<Image>().color.a > 0) // continue until fully transparent
        {
            fadeAmount = objectColor.a - (fadeSpeed * Time.deltaTime); // how much the thing should fade by
            objectColor = new Color(objectColor.r, objectColor.g, objectColor.b, fadeAmount); // change colour by fade amount
            blackScreen.GetComponent<Image>().color = objectColor; // set this colour to the screen
            yield return null; // stop code when completed loop
        }
    }
}
```

I copy-pasted the fade-to-black code, reversing the if statement, making sure it went until transparent, and reversing the fade amount. I also slightly edited the Coroutine's name for posterity. The next thing to

do would be to add the reference in the inspector panel and to call this coroutine in playerDeath, in player. I also edited the name of the script to FadeUI, to better reflect its purpose (so it can be reused in other elements if necessary).

```
2 references
private IEnumerator playerDeath()
{
    blackScreen.GetComponent<FadeToBlack>().FadeBlackScreen(true);
    yield return new WaitForSeconds(blackScreen.GetComponent<FadeToBlack>().getFadeSpeed());
    transform.position = new Vector3(0, 0, 0);
    blackScreen.GetComponent<FadeToBlack>().FadeBlackScreen(false);
}
```

I decided to make playerDeath a coroutine to add delays between fading to black and switching position. In order to do the above, I realised I needed to get the fade time to delay for the same amount of time it would take to fade to black, so I had to add a getter (or accessor) into the FadeUI script. This code will be changed in later stages, with the vector being replaced by the correct respawn point and a pause being added while switching screens so that the user can't move around while dead/respawning.

```
UnassignedReferenceException: The variable blackScreen of Player has not been assigned.
```

The above error was shown as I attempted to run the code – I'd forgotten to add the reference to the black screen to the player. I added it in the inspector panel and tried again. No error was returned, but the black screen wasn't fading in or out, even if the teleportation was occurring.

```
2 references
private IEnumerator playerDeath()
{
    StartCoroutine(blackScreen.GetComponent<FadeToBlack>().FadeBlackScreen(true));
    yield return new WaitForSeconds(blackScreen.GetComponent<FadeToBlack>().getFadeSpeed());
    transform.position = new Vector3(0, 0, 0);
    StartCoroutine(blackScreen.GetComponent<FadeToBlack>().FadeBlackScreen(false));
}
```

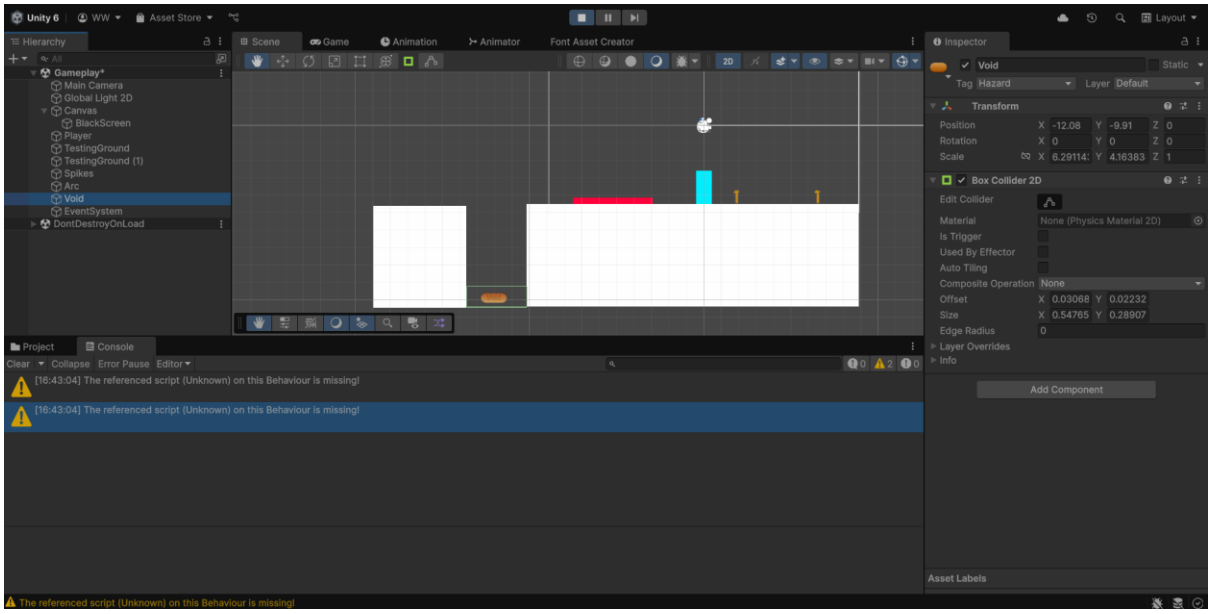
The issue was that I wasn't actually starting the coroutines, merely getting their references. I then ran into an issue of timing, where it was taking too long on the full black screen. I also realised that the user could collide with a hazard when already 'dead' if they hadn't yet teleported.

```
0 references
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag(GROUND_TAG))
    {
        isGrounded = true;
        doubleJump = true;
    }
    if (collision.gameObject.CompareTag(HAZARD_TAG) && isAlive)
    {
        isAlive = false;
        StartCoroutine(playerDeath());
    }
}
```

```
0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag(HAZARD_TAG) && isAlive)
    {
        StartCoroutine(playerDeath());
    }
}
```

```
2 references
private IEnumerator playerDeath()
{
    StartCoroutine(blackScreen.GetComponent<FadeToBlack>().FadeBlackScreen(true));
    yield return new WaitForSeconds(1.5f);
    transform.position = new Vector3(0, 0, 0);
    yield return new WaitForSeconds(0.5f);
    StartCoroutine(blackScreen.GetComponent<FadeToBlack>().FadeBlackScreen(false));
    isAlive = true;
}
```

I added a Boolean qualifier to ensure that the player was alive, so the coroutine couldn't be called multiple times. I also set the time between fading in and out to a static 2 seconds, with a break 1.5 seconds in to teleport the player (so it wasn't still falling when fading out). This was acceptable



Finally, I added the void. I had to increase the size and number of the platforms to create a void. It is an empty object with a collider (so the player doesn't fall too far and to measure collision. After running, no issues occurred and the player was able to be killed. The camera limit established in stage 1 worked well enough – I will re-assess when creating the levels to see if a more detailed style of limit is necessary. The console has been returning the warning seen above, but it does not seem to be causing any problems, nor can I really tell where it is coming from. Thus, I have made a note of it but decided to ignore for now, and will bear this in mind if any unforeseen errors occur.

```
0 references
public class Cannonball : MonoBehaviour
{
    0 references
    void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Ground") || collision.CompareTag("Hazard"))
            Destroy(gameObject);
    }
}
```

This is a response to the failed test 9. I added this into the cannonball script, so it would be destroyed when hitting a hazard – although it doesn't make much difference for the spikes (because it would hit the ground beneath the spikes anyway. This only really matters with the void, as otherwise it will go through and just keep going.

Testing

Test No.	Description	Type	Data	Expected Result	Actual Result
1	'Spikes' sprite shown	Valid	Run program	Spikes are visible on the screen	Stage 2 (1).mp4 00:00
2	Spike hitbox	Valid	Character on spikes	Spikes are solid objects that the character can walk on	Stage 2 (1).mp4 00:02
3	Spikes can kill	Valid	Character touches spikes	Character sprite is teleported to (0,0,0)	Stage 2 (1).mp4 00:03
4	Transition functions	Valid	Character touches spikes	Screen fades to black, then fades back to gameplay with	Stage 2 (1).mp4 00:04

				character sprite teleported to (0,0,0)	
5	'Arc' sprite shown	Valid	Run program	Electricity arc is visible on screen	Stage 2 (1).mp4 00:00
6	Arc has no hitbox	Invalid	Character inside electricity	Character falls through the arc	Stage 2 (1).mp4 00:08
7	Arc can kill	Valid	Character touches arc	Death transition occurs etc.	Stage 2 (1).mp4 00:11
8	Void can kill	Valid	Character touches void	Death transition occurs	Stage 2 (1).mp4 00:21
9a	Spikes destroy cannonball	Valid	Cannonball touches spikes	Cannonball is destroyed	Stage 2 (1).mp4 00:17
9b	Arc does not destroy cannonball	Invalid	Cannonball touches arc	Cannonball goes through arc and is destroyed by ground	Stage 2 (2).mp4 00:02
9c	Void destroys cannonball	Valid	Cannonball touches void	Cannonball is destroyed	Stage 2 (2).mp4 00:05
9c	Void destroys cannonball	Valid	Cannonball touches void	Cannonball is destroyed	Stage 2 (3).mp4 00:02
10	Camera cannot pan down to see character on void	Valid	Character falls onto void	Camera does not move downwards as character falls into void	Stage 2 (1).mp4 00:20

Evaluation

My aims for this stage were;

- show placeholder sprites for hazards
- give hitboxes to the hazards (not arc)
- 'kill' character on touch
- reset character to beginning
- transition from death to respawn (black screen fade)
- create 'void' hazard
- ensure cannonball is destroyed on contact with hazards
- code some preliminary camera limits

Most of these were fulfilled to an extent that will remain in the final game. As with all things, the sprites will be retouched in Stage 12, but most code will not change. The camera limitations are the only potential failure of this stage, not being quite as fluid as I want them to be. For the sake of meeting deadlines I will continue to make new things, next addressing them in Stage 8.

Stage 3 – Checkpoints

This stage heavily relies on the previous stage. Building on the respawn functionality added in the last, checkpoints that serve as places to respawn are added. These must be interacted with in order to set respawn point, where the player will reappear if they touch one of the hazards recently introduced. I need to ensure that respawn points overwrite the position of the last one used, and that they function as intended (only if interacted with). In addition, to show that they need to be interacted with a pop-up showing the key-bind for interacting will appear when the checkpoint is touched.

Functionality:

- checkpoint sprite
- interact pop-up
- sprite change when interacted with
- respawn position changes on interaction

Design

Data Dictionary

NAME	TYPE	Class	Parameters	DESCRIPTION
Awake()	Method	Checkpoint		Run on initialisation
Update()	Method	Checkpoint		Run every frame
checkpointPosition	Vector3	Checkpoint		Position of checkpoint
playerPosition	Vector3	Checkpoint		Position of player
activationDistance	float	Checkpoint		Max. distance the player must be away from the checkpoint for it to work and for the pop-up to show
popup()	void	Checkpoint	bool show	Method that shows or hides the interact popup. Changed to IEnumerator.
activeCheckpoint	bool	Checkpoint		T/F value determining if it is currently being used
enteredRange	bool	Checkpoint		Determines whether the player has already entered the range of the checkpoint. Added later.
animator	Animator	Checkpoint		Reference to the animator
respawnPosition	Vector3	Player		Where the player will respawn
setRespawnPosition()	Void	Player	Vector3 newPosition	Setter for respawnPosition
getRespawnPosition()	Vector3	Player		Getter for respawnPosition

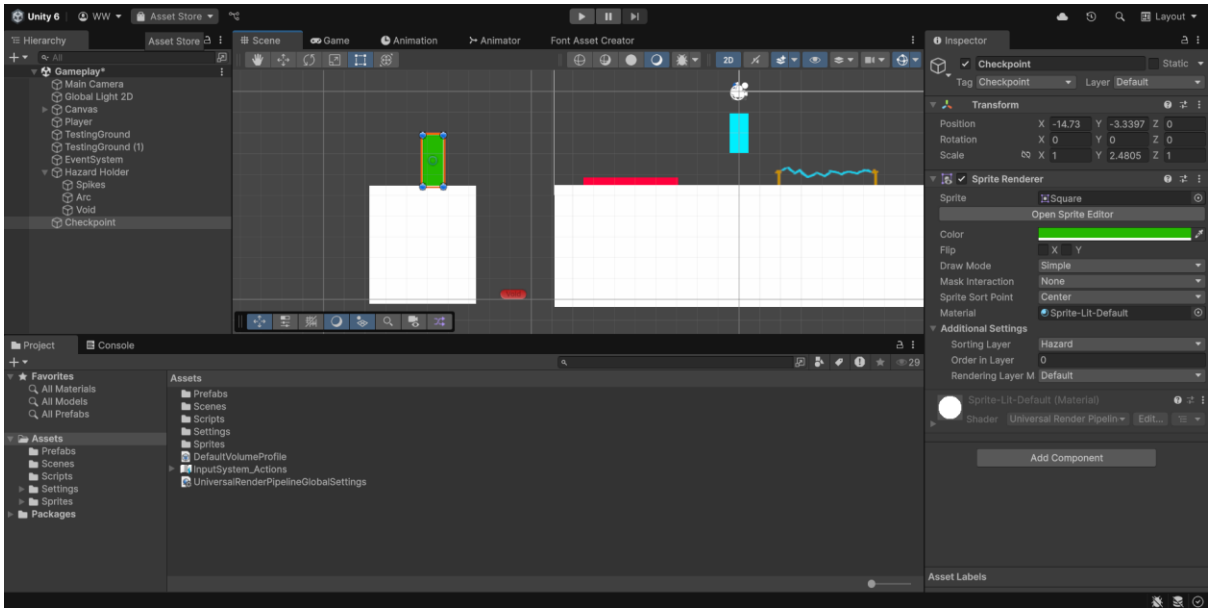
Pseudocode

```
CLASS Player
    PRIVATE VECTOR3 respawnPoint = (0,0,0)

    PROCEDURE setRespawnPoint(VECTOR3 newPoint)
        respawnPoint = newPoint
    ENDPROCEDURE

    PROCEDURE getRespawnPoint()
        RETURN respawnPoint
    ENDPROCEDURE
ENDCLASS
```

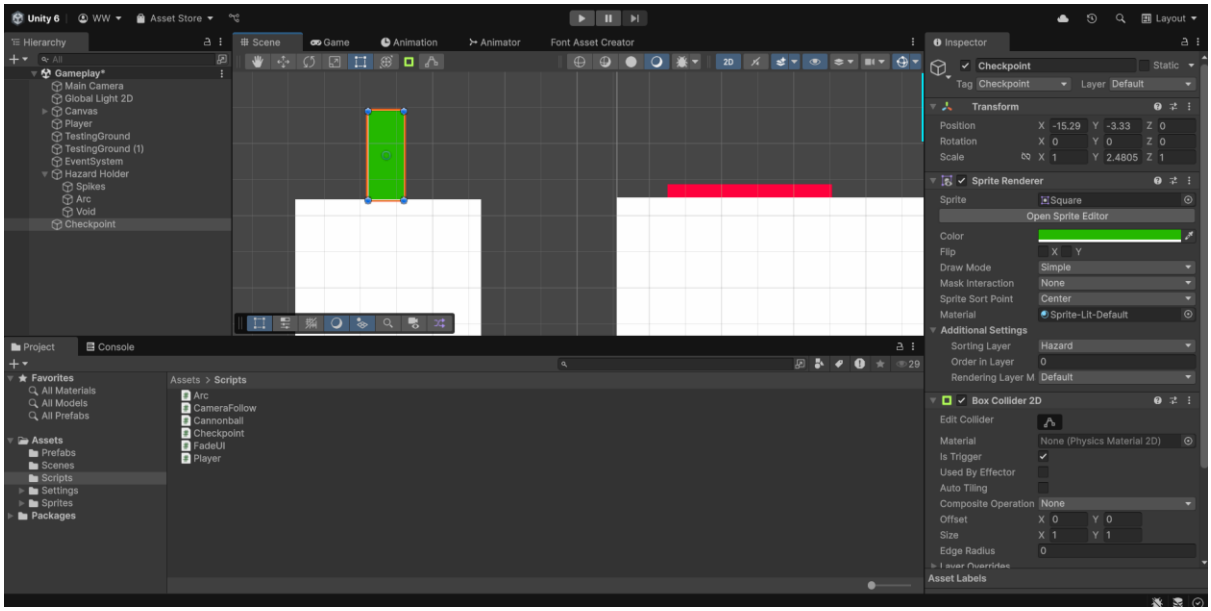
^added to Player script

```
CLASS Checkpoint
    PRIVATE VECTOR3 checkpointPosition
    PRIVATE VECTOR3 playerPosition
    PRIVATE FLOAT activationDistance
    PRIVATE BOOLEAN activeCheckpoint
    PRIVATE ANIMATOR animator

    PROCEDURE Awake()
        checkpointPosition = this.POSITION
        animator = this.ANIMATOR
    ENDPROCEDURE

    PROCEDURE Update()
        playerPosition = player.position
        float distance = distanceBetween(checkpointPosition, playerPosition)
        IF distance < activationDistance THEN
            popup(TRUE)
            IF keyPressed(E) OR keyPressed(F) THEN
                activeCheckpoint = TRUE
                animator.changeState
            ENDIF
        ELSE
            popup(FALSE)
        ENDIF
        IF activeCheckpoint AND player.checkpoint != activeCheckpoint THEN
            activeCheckpoint = FALSE
            animator.changeState()
        ENDIF
    ENDPROCEDURE

    PROCEDURE popup(BOOLEAN show)
        IF show THEN
            popup.POSITION.X += 1
            IF popup.POSITION.X > maxPopup THEN
                popup.POSITION.X = maxPopup
            ENDIF
        ELSE
            popup.POSITION.X -= 1
            IF popup.POSITION.X < minPopup THEN
                popup.POSITION.X = minPopup
            ENDIF
        ENDIF
    ENDPROCEDURE
ENDCLASS
```

Development



The first thing to do was to add the checkpoint. I assigned it a very basic placeholder sprite, the Checkpoint tag, and put it in a sorting layer below the ground so that it would look more natural.



I added a box collider, ensuring that it has IsTrigger ticked – this is because the checkpoint is not tangible, and the player should be able to stand in front of it and use it.

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

0 references
public class Checkpoint : MonoBehaviour
{
    2 references
    private Vector3 checkpointPosition; // position of checkpoint
    2 references
    private Vector3 playerPosition; // postition of player
    1 reference
    [SerializeField] float activationDistance; // distance that the checkpoint will activate from

    0 references
    void Start()
    {
        checkpointPosition = transform.position;
    }

    0 references
    void Update()
    {
        playerPosition = GameObject.FindWithTag("Player").transform.position; // gets the position of the player
        float distance = Vector3.Distance(checkpointPosition, playerPosition); // distance between the two
        if (distance < activationDistance)
        {
            // show pop-up for Interact
            if (Input.GetKeyDown("E") || Input.GetKeyDown("F")) // if press E or F
            {
                // remove pop up
                // do functions
            }
        }
    }
}
```

This is the outline of the checkpoint script so far. The positions it holds are to find the distance of the player from the checkpoint, which will allow it to be used and create the bound where the pop up is shown. This is to increase accessibility, so as to not assume that users know what to do, and comply with the stakeholder recommendation that tutorials should be in the form of assistance throughout, rather than a dedicated tutorial level. The input keys are E or F, as shown in the Analysis stage.

```
private Vector3 respawnPosition = new Vector3(0, 0, 0); // respawn position (accessed by Checkpoint)
```

```
1 reference
public void setRespawnPosition(Vector3 newPosition)
{
    respawnPosition = newPosition;
}
```

```
2 references
private IEnumerator playerDeath()
{
    StartCoroutine(blackScreen.GetComponent<FadeToBlack>().FadeBlackScreen(true));
    yield return new WaitForSeconds(1.5f);
    transform.position = respawnPosition;
    yield return new WaitForSeconds(0.5f);
    StartCoroutine(blackScreen.GetComponent<FadeToBlack>().FadeBlackScreen(false));
    isAlive = true;
}
```

I created a new attribute in the player, with default value (0, 0, 0), which represents the position at which the player will respawn. This will also be used when saving and loading. I decided to make it a private variable because this encapsulation increases security and ensures I will not accidentally change it. I also briefly edited the playerDeath() function to respawn at this position.

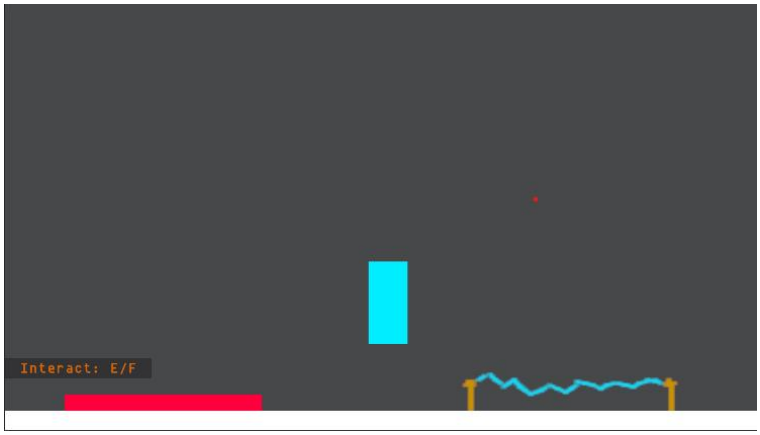
```
0 references
void Update()
{
    playerPosition = GameObject.FindWithTag("Player").transform.position; // gets the position of the player
    float distance = Vector3.Distance(checkpointPosition, playerPosition); // distance between the two
    if (distance < activationDistance)
    {
        // show pop-up for Interact
        if (Input.GetKeyDown("E") || Input.GetKeyDown("F")) // if press E or F
        {
            // remove pop up
            GameObject.FindWithTag("Player").GetComponent<Player>().setRespawnPosition(checkpointPosition); // set position to
        }
    }
}
```

This sets the respawn position to the correct location, using the setter in Player.

```
1 reference
public Vector3 getRespawnPosition()
{
    return respawnPosition;
}
```

```
void Update()
{
    playerPosition = GameObject.FindWithTag("Player").transform.position; // gets the position of the player
    float distance = Vector3.Distance(checkpointPosition, playerPosition); // distance between the two
    if (distance < activationDistance)
    {
        // show pop-up for Interact
        if (Input.GetKeyDown("E") || Input.GetKeyDown("F")) // if press E or F
        {
            // remove pop up
            GameObject.FindWithTag("Player").GetComponent<Player>().setRespawnPosition(checkpointPosition); // set position to
            activeCheckpoint = true;
        }
    }
    Vector3 currentCheckpoint = GameObject.FindWithTag("Player").GetComponent<Player>().getRespawnPosition(); // get current re
    if (currentCheckpoint != checkpointPosition)
    {
        activeCheckpoint = false; // ANIMATION STUFF WILL GO HERE
    }
}
```

I then laid the groundwork for the only other animation I definitely wanted to include – the indicator that a checkpoint is in use or not. This will be developed later in the stage, as functionality is more important, but felt like a good time to add (as it is fundamentally very similar to setting the respawn position).



I then created a pop-up prompt. It is slightly transparent, so as not to blend in with the game, and is down in the bottom right so it is not overly intrusive.

```
0 references
void Update()
{
    playerPosition = GameObject.FindWithTag("Player").transform.position; // gets the position of the player
    float distance = Vector3.Distance(checkpointPosition, playerPosition); // distance between the two
    if (distance < activationDistance)
    {
        if (!activeCheckpoint) // if not active checkpoint
        {
            popup(true); // show popUp
        }
        if (Input.GetKeyDown("E") || Input.GetKeyDown("F")) // if press E or F
        {
            popup(false); // remove popUp
            GameObject.FindWithTag("Player").GetComponent<Player>().setRespawnPosition(checkpointPosition); // set position to
            activeCheckpoint = true;
        }
    }
    Vector3 currentCheckpoint = GameObject.FindWithTag("Player").GetComponent<Player>().getRespawnPosition(); // get current re
    if (currentCheckpoint != checkpointPosition)
    {
        activeCheckpoint = false; // ANIMATION STUFF WILL GO HERE
    }
}
```

```
2 references
void popup(bool show)
{
    if (show)
    {
        while (interactPopUp.transform.position.x < -776f)
        {
            interactPopUp.transform.position += new Vector3(5f, 0f, 0f);
            if (interactPopUp.transform.position.x > -776f)
            {
                interactPopUp.transform.position = new Vector3(-776f, interactPopUp.transform.position.y, 0f);
            }
        }
    }
    else
    {
        while (interactPopUp.transform.position.x < -1146f)
        {
            interactPopUp.transform.position -= new Vector3(5f, 0f, 0f);
            if (interactPopUp.transform.position.x < -1146f)
            {
                interactPopUp.transform.position = new Vector3(-1146f, interactPopUp.transform.position.y, 0f);
            }
        }
    }
}
```

I added the popup method, which should serve to manipulate the appearance and disappearance of the popup. I made it more complex than simply appearing and disappearing to hopefully draw the eye and

look a little better. In addition, I made the popup disappear when the checkpoint is active, as it does not need to be activated after that.

```
ArgumentException: Input Key named: E is unknown
if (Input.GetKeyDown(KeyCode.E) || Input.GetKeyDown(KeyCode.F)) // if press E or F
```

Upon running, this error was shown. I needed to identify the actual keycode of E and F. By being this specific, I am limiting the programs cross-functionality on other devices, especially ones without keyboard. It would be more generally sustainable to create an "Interact" key, which is different depending on input devices, but it is currently needlessly complex as I have no plans to release on any devices without a keyboard.

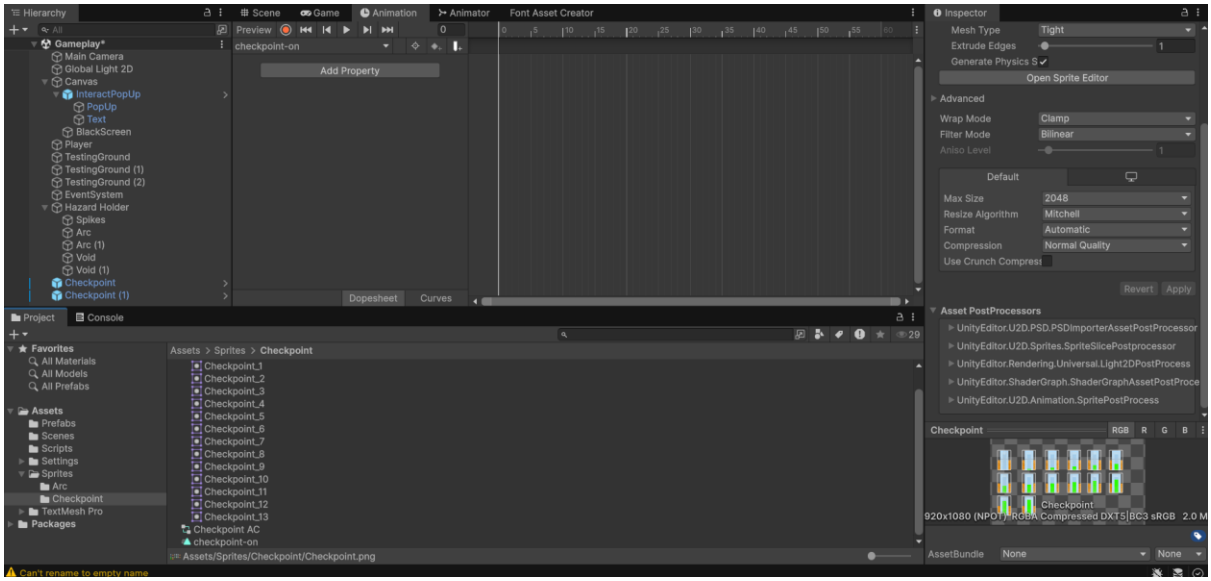
After running the code, the editor was crashing whenever I exited the range of the checkpoint. I believe that this was due to memory constraints, where the while loops in the Checkpoint script were overwhelming the memory.

```
4 references
IEnumerator popup(bool show)
{
    if (show)
    {
        Debug.Log("showing");
        while (interactPopUp.transform.position.x < 960f)
        {
            interactPopUp.transform.position += new Vector3(5f, 0f, 0f);
            if (interactPopUp.transform.position.x > 960f)
            {
                interactPopUp.transform.position = new Vector3(960f, interactPopUp.transform.position.y, 0f);
                yield return null;
            }
        }
    }
    else
    {
        while (interactPopUp.transform.position.x > 590f)
        {
            interactPopUp.transform.position -= new Vector3(5f, 0f, 0f);
            if (interactPopUp.transform.position.x < 590f)
            {
                interactPopUp.transform.position = new Vector3(590f, interactPopUp.transform.position.y, 0f);
                yield return null;
            }
        }
    }
}
```

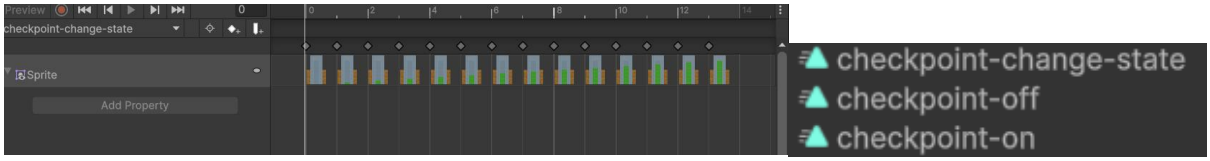
The first thing to do was turn the popup function into a coroutine, so that it would execute fully a single time and then stop. This removes the multiple while loops stacking up when exiting (as it tries to show and hide simultaneously), as I can start and stop the coroutine at will.

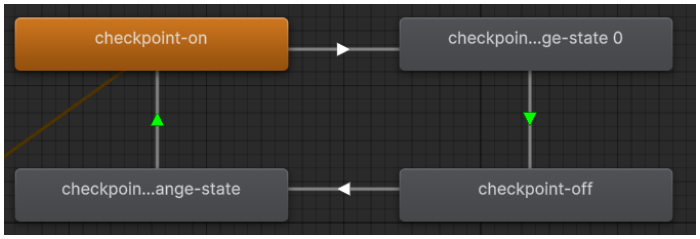
```
playerPosition = GameObject.FindWithTag("Player").transform.position; // gets the position of the player
float distance = Vector3.Distance(checkpointPosition, playerPosition); // distance between the two
if (distance < activationDistance)
{
    enteredRange = true;
    if (!activeCheckpoint) // if not active checkpoint
    {
        StopCoroutine(popup(false)); // stop any coroutine hiding the popup
        StartCoroutine(popup(true)); // start showing the popup
    }
    if (Input.GetKeyDown(KeyCode.E) || Input.GetKeyDown(KeyCode.F)) // if press E or F
    {
        StopCoroutine(popup(true)); // stop any coroutine showing the popup
        StartCoroutine(popup(false)); // start hiding the popup
        popup(false); // remove popUp
        GameObject.FindWithTag("Player").GetComponent<Player>().setRespawnPosition(checkpointPosition); // set position to
        activeCheckpoint = true;
        // ANIMATION STUFF GOES HERE
    }
}
else
{
    if (enteredRange) // if has been within range and then exited
    {
        enteredRange = false; // reset
        Debug.Log("Exit");
        StopCoroutine(popup(true)); // stop any coroutine showing the popup
        StartCoroutine(popup(false)); // start hiding the popup
    }
}
```

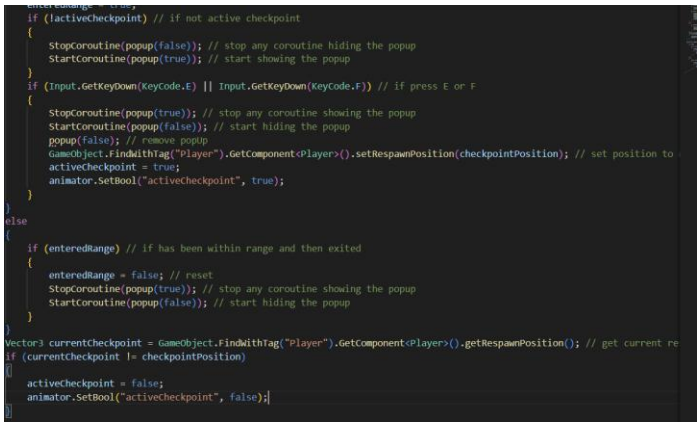
I replaced the previous function with this. Using the new Boolean value for enteredRange, I could determine when the player had just exited the range and call the hide function then. I replaced any call with stopping the opposite coroutine and starting the necessary, to make sure two things didn't happen simultaneously and potentially break the game.



Starting on the animation work, I added the custom-made sprite sheet and an animator controller, assigning the latter to the Animator on the Checkpoint. I created an animation, to which I will add the 'on' sequence.







```
if (!activeCheckpoint) // if not active checkpoint
{
    StopCoroutine(popup(false)); // stop any coroutine hiding the popup
    StartCoroutine(popup(true)); // start showing the popup
}
if (Input.GetKeyDown(KeyCode.E) || Input.GetKeyDown(KeyCode.F)) // if press E or F
{
    StopCoroutine(popup(true)); // stop any coroutine showing the popup
    StartCoroutine(popup(false)); // start hiding the popup
    popup(false); // remove popUp
    GameObject.FindWithTag("Player").GetComponent<Player>().setRespawnPosition(checkpointPosition); // set position to
    activeCheckpoint = true;
    animator.SetBool("activeCheckpoint", true);
}
}
else
{
    if (enteredRange) // if has been within range and then exited
    {
        enteredRange = false; // reset
        StopCoroutine(popup(true)); // stop any coroutine showing the popup
        StartCoroutine(popup(false)); // start hiding the popup
    }
}
Vector3 currentCheckpoint = GameObject.FindWithTag("Player").GetComponent<Player>().getRespawnPosition(); // get current re
if (currentCheckpoint != checkpointPosition)
{
    activeCheckpoint = false;
    animator.SetBool("activeCheckpoint", false);
}
```

I created three animations – both on and off states are just the initial/final sprite, and the change=state animation is used to transfer between them (reversed when turning off). When the script changes the activeCheckpoint Boolean in the animator to false, it runs the reversed change-state animation and then loops on the off animation, and vice versa.

Testing

Test No.	Description	Type	Data	Expected Result	Actual Result
1	Checkpoint sprite shown	Valid	Run program	Checkpoint sprite is visible on screen	Stage 3.mp4 00:00
2	Checkpoint has no hitbox	Invalid	Run program	Checkpoint cannot be stood on	Stage 3.mp4 00:00
3	Interact pop-up appears	Valid	Character touches checkpoint	Pop-up appears on screen	Stage 3.mp4 00:00 Stage 3.mp4 00:03
4	Alt. Checkpoint sprite shown	Valid	Run program with Active sprite	Alt. Checkpoint sprite is visible on screen	Stage 3.mp4 00:03
5a	‘F’ key works	Valid	‘F’ pressed while touching checkpoint	Checkpoint switches sprites	Stage 3.mp4 00:04
5b	‘E’ key works	Valid	‘E’ pressed while touching checkpoint	Checkpoint switches sprites	Stage 3.mp4 00:06
6a	Respawn works	Valid	Character touches hazard after checkpoint interaction	Respawns at interacted checkpoint	Stage 3.mp4 00:12
6b	Respawn overwrite works	Valid	Character touches hazard after interacting with two distinct checkpoints	Respawns at second checkpoint	Stage 3.mp4 00:18

Evaluation

My aims in this stage were as follows:

- checkpoint sprite

- interact pop-up
- sprite change when interacted with
- respawn position changes on interaction

I think these were all fully fulfilled, and I am unlikely to change them over the course of the development. This means I have a very strong foundation on which to build in Stage 6. In addition, the changing of the code that was necessary (due to crashing) has made the run much smoother than it would otherwise be, so I believe that this stage is very successful.

Stage 4 – Basic Enemy

An enemy is the next step up from a hazard. I chose to program this after the checkpoints were added because enemies need to respawn after the player is killed to maintain challenge level in an area. These enemies must activate when the player comes within a certain distance, follow the player for some distance and be able to traverse the environment. If the above does not occur, they are effectively hazards. To make them a threat, they should be able to do an attack that harms the player at melee range (so that the player can dodge and counter more easily). The player's cannonball needs to be able to destroy the enemies so that the threat can be removed, unlike with hazards, to make sure that the cannonball is an effective weapon.

Functionality:

- display enemy sprite
- create enemy spawn point
- spawn enemies when the player gets close enough
- enemy movement (being able to get up small hills and ledges)
- enemies go towards the player if the environment is traversable
- enemies follow for a certain distance from spawn points
- enemies return to spawn points when player leaves range
- enemies despawn when player gets far enough away
- enemies respawn when player dies
- enemies attack player at close range
- enemies are destroyed by cannonballs (and despawn cannonballs when hit)

Design

Data Dictionary

NAME	TYPE	Class	Parameters	DESCRIPTION
Awake()	void	Enemy, EnemySpawner, GroundDetector, JumpDetector		Executes at start of game
Update()	void	EnemySpawner		Executes every frame
FixedUpdate()	void	Enemy		Executes at a fixed interval of time, used for physics calculations
EnemyDeath()	void	Enemy		Removes enemy
setMoveSpeedSign()	void	Enemy	bool positive	Sets the sign for the moveSpeed attribute to positive or negative
reachedEdge()	void	Enemy		Called when edge is reached, sets a movement cap at that position

jumpLedge()	void	Enemy		Jumps when notified that it is at a wall/ledge
moveSpeed	float	Enemy		Speed of enemy movement
chaseDistance	Float	Enemy		How close the player must be for the enemy to begin moving towards it
player	GameObject	Enemy		Holds reference to player
body	Rigidbody2D	Enemy		Holds reference to Rigidbody2D
moveCapLeft	float	Enemy		x-coordinate at which the enemy cannot move past (left)
moveCapRight	float	Enemy		x-coordinate at which the enemy cannot move past (right)
spawnpoint	Vector3	Enemy		Where the enemy spawned
jumping	bool	Enemy		If the enemy is jumping
ledgeBelowLeft	bool	Enemy		If there is a ledge below and to the left of the enemy to fall on. Added in development
ledgeBelowRight	bool	Enemy		If there is a ledge below and to the right of the enemy to fall on. Added in development
spawnEnemy()	void	EnemySpawner		Instantiates enemy object
resetSpawner()	void	EnemySpawner		Resets spawner on player death/moving out of range
OnEnable()	void	EnemySpawner		Subscribes to player's death event
OnDisable()	void	EnemySpawner		Unsubscribes from the above
enemy	GameObject	EnemySpawner		Enemy that will be instantiated
spawnedEnemy	GameObject	EnemySpawner		Reference to the actual spawned enemy, not the prefab. Added in development
enemySpawned	bool	EnemySpawner		Records whether the spawner has already spawned an enemy
spawnerPosition	Vector3	EnemySpawner		Position of spawner
playerPosition	Vector3	EnemySpawner		Position of player
renderDistance	float	EnemySpawner		Distance from the player in which the enemy is spawned
despawnDistance	float	EnemySpawner		Distance from the player beyond which the enemy is despawned
OnTriggerExit2D()	void	GroundDetector	Collider2D collision	Used when an IsTrigger collider exits another collider
OnTriggerEnter2D()	void	GroundDetector, JumpDetector	Collider2D collision	Used when an IsTrigger collider enters another collider
parentEnemy	GameObject	GroundDetector, JumpDetector		Holds the reference to the parent enemy of a detector
OnTriggerExit2D()	void	LowerLedge-LeftDetector, LowerLedge-RightDetector	Collider2D collision	Used when an IsTrigger collider exits another collider
OnTriggerStay2D()	void	LowerLedge-LeftDetector, LowerLedge-RightDetector	Collider2D collision	Used every frame an IsTrigger collider remains within a collider
parentEnemy	GameObject	LowerLedge-LeftDetector,		Holds the reference to the parent enemy

		LowerLedge-RightDetector		
ledge	bool	LowerLedge-LeftDetector, LowerLedge-RightDetector		Records whether or not there is a ledge below the enemy (in the requisite direction). Entire scripts were added in development

Pseudocode

```
CLASS Enemy
    PRIVATE FLOAT moveSpeed = 3
    PRIVATE FLOAT chaseDistance = 7
    PRIVATE GAMEOBJECT player
    PRIVATE RIGIDBODY2D body
    PRIVATE FLOAT moveCapLeft
    PRIVATE FLOAT moveCapRight
    PUBLIC VECTOR3 spawnpoint
    PUBLIC BOOLEAN jumping

    PROCEDURE Awake()
        player = GET(Player)
        body = this.RIGIDBODY2D
    ENDPROCEDURE

    PROCEDURE EnemyDeath()
        DESTROY(this)
    ENDPROCEDURE

    PROCEDURE FixedUpdate()
        FLOAT distance = distanceBetween(this.position, player.position)
        IF distance < chaseDistance THEN
            IF player.x < this.x AND this.x > moveCapLeft THEN
                setMoveSpeedSign(FALSE)
                body.FORCE(moveSpeed)
            ELSE IF player.x > this.x AND this.x < moveCapRight THEN
                setMoveSpeedSign(TRUE)
                body.FORCE(moveSpeed
            ENDIF
        ELSE
            IF spawnpoint.x < this.x  THEN
                setMoveSpeedSign(FALSE)
                body.FORCE(moveSpeed)
            ELSE IF spawnpoint.x > this.x THEN
                setMoveSpeedSign(TRUE)
                body.FORCE(moveSpeed
            ENDIF
        ENDIF
    ENDPROCEDURE

    PROCEDURE setMoveSpeedSign(BOOLEAN positive)
        IF positive THEN
            IF moveSpeed < 0 THEN
                moveSpeed = -moveSpeed
            ENDIF
        ELSE
            IF moveSpeed > 0 THEN
                moveSpeed = -moveSpeed
            ENDIF
        ENDIF
    ENDPROCEDURE

    PUBLIC PROCEDURE reachedEdge()
        IF !jumping THEN
            IF moveSpeed < 0 THEN
                moveCapLeft = this.x
            ELSE
                moveCapRight = this.x
            ENDIF
        ENDIF
    ENDPROCEDURE
ENDCLASS
```

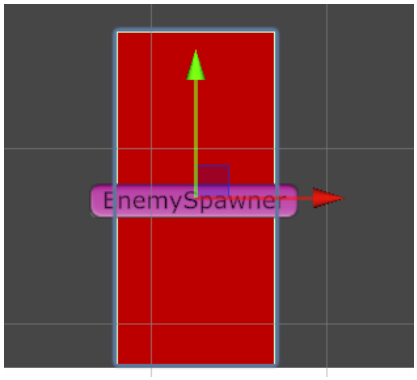
```
CLASS EnemySpawner
    PRIVATE GAMEOBJECT enemy
    PRIVATE BOOLEAN enemySpawned = FALSE
    PRIVATE VECTOR3 spawnerPosition
    PRIVATE VECTOR3 playerPosition
    PRIVATE FLOAT renderDistance = 10
    PRIVATE FLOAT despawnDistance = 25

    PROCEDURE Awake()
        spawnerPosition = this.position
    ENDPROCEDURE

    PROCEDURE Update()
        playerPosition = GET(player).position
        FLOAT distance = distanceBetween(spawnerPosition, playerPosition)
        IF distance < renderDistance AND !enemySpawned THEN
            spawnEnemy()
        ENDIF
        IF distance > despawnDistance AND enemySpawned THEN
            resetSpawner()
        ENDIF
    ENDPROCEDURE

    PROCEDURE spawnEnemy()
        enemySpawned = TRUE
        CREATE(enemy)
        enemy.spawnpoint = this.position
    ENDPROCEDURE

    PROCEDURE resetSpawner()
        IF enemy != NULL THEN
            DESTROY(enemy)
        ENDIF
        enemySpawned = FALSE
    ENDPROCEDURE

    PROCEDURE onEnable()
        SUBSCRIBE(Player.onDeath)
    ENDPROCEDURE

    PROCEDURE onDisable()
        UNSUBSCRIBE(Player.onDeath)
    ENDPROCEDURE
ENDCLASS
```

```
CLASS GroundDetector
    PRIVATE GAMEOBJECT parentEnemy

    PROCEDURE Awake()
        parentEnemy = this.parent
    ENDPROCEDURE

    PROCEDURE OnTriggerEnter2D(COLLIDER2D collision)
        IF collision = "Ground" THEN
            parentEnemy.reachedEdge()
        ENDIF
    ENDPROCEDURE

    PROCEDURE OnTriggerExit2D(COLLIDER2D collision)
        IF collision = "Ground" THEN
            parentEnemy.jumping = TRUE
        ENDIF
    ENDPROCEDURE
ENDCLASS
```

```
CLASS JumpDetector
    PRIVATE GAMEOBJECT parentEnemy

    PROCEDURE Awake()
        parentEnemy = this.parent
    ENDPROCEDURE

    PROCEDURE OnTriggerEnter2D(COLLIDER2D collision)
        IF collision = "Ground" THEN
            parentEnemy.jumpLedge()
        ENDIF
    ENDPROCEDURE
ENDCLASS
```

Also add event creation into player (too small and specific to pseudocode).

Development



First, I created the placeholder for the enemy, and put it on its own platform. I gave it a Rigidbody and a Collider to make it act as a physical object, and placed it in the same sorting layer as the player. Every enemy will have a corresponding spawner, which will instantiate that enemy when the player comes within a certain distance and reset when the player respawns. I assigned the Enemy object with the corresponding tag, which will be used to determine collisions.

```
public class EnemySpawner : MonoBehaviour
{
    1 reference
    [SerializeField] GameObject enemy;
    2 references
    bool enemySpawned = true;
    1 reference
    [SerializeField] float renderDistance;
    2 references
    Vector3 playerPosition;
    2 references
    Vector3 spawnerPosition;

    0 references
    void Awake()
    {
        spawnerPosition = transform.position;
    }

    0 references
    void Update()
    {
        playerPosition = GameObject.FindWithTag("Player").transform.position; // gets the position of the player
        float distance = Vector3.Distance(spawnerPosition, playerPosition); // distance between the two
        if (distance < renderDistance && !enemySpawned)
        {
            spawnEnemy();
        }
    }


    1 reference
    private void spawnEnemy()
    {
        enemySpawned = true;
        Instantiate(enemy);
    }
}
```

I created the basic structure of the spawner's script. I could reuse the distance code from checkpoint, ensuring that I changed variable names to maintain the integrity of the code. At the moment, the enemy spawner does not reset when the player dies.

```
1 reference
private void spawnEnemy()
{
    enemySpawned = true;
    spawnedEnemy = Instantiate(enemy);
    spawnedEnemy.transform.position = spawnerPosition;
}

3 references
private void resetSpawner()
{
    if (spawnedEnemy != null)
        spawnedEnemy.GetComponent<Enemy>().EnemyDeath();
    enemySpawned = false;
}
```

The spawnEnemy function must be replaced as follows, in order to get a reference to the object created, not the prefab. If not, it will be unable to delete. This ensures that the resetSpawner function still works even if the spawnedEnemy is dead by checking for null.

```
2 references
private IEnumerator playerDeath()
{
    StartCoroutine(blackScreen.GetComponent<FadeToBlack>().FadeBlackScreen(true));
    yield return new WaitForSeconds(1.5f);
    transform.position = respawnPosition;
    yield return new WaitForSeconds(0.5f);
    StartCoroutine(blackScreen.GetComponent<FadeToBlack>().FadeBlackScreen(false));
    onDeath?.Invoke();
    isAlive = true;
}

1 reference
public delegate void OnDeath();
3 references
public static event OnDeath onDeath;
```

```
2 references
private void resetSpawner()
{
    if (enemy != null)
        enemy.GetComponent<Enemy>().EnemyDeath(); // run code in enemy which will destroy itself
    enemySpawned = false;
}

0 references
private void OnEnable()
{
    Player.onDeath += resetSpawner; // subscribe to onDeath
}
0 references
private void OnDisable()
{
    Player.onDeath -= resetSpawner; // unsubscribe from onDeath
}
```

This event will be subscribed to by the enemy spawners, and is called when the player dies. The resetSpawner doesn't directly destroy the enemy it is linked to, but calls a function in the enemy – this allows for animations if necessary, and anything else that may occur on an enemy's death. It then allows the spawner to spawn another enemy. This is most of the functionality of the spawner itself, except for the despawning of enemies when the player gets far enough away (to reduce load).

```
1 reference
[SerializeField] float despawnDistance;
2 references
Vector3 playerPosition;
3 references
Vector3 spawnerPosition;

0 references
void Awake()
{
    spawnerPosition = transform.position;
}

0 references
void Update()
{
    playerPosition = GameObject.FindWithTag("Player").transform.position; // gets the position of the player
    float distance = Vector3.Distance(spawnerPosition, playerPosition); // distance between the two
    if (distance < renderDistance && !enemySpawned)
    {
        spawnEnemy();
    }
    if (distance > despawnDistance && enemySpawned)
    {
        resetSpawner();
    }
}
```

When the player gets far enough away, the spawner will reset (similar to when the player dies). This ensures that if enemies are not defeated, they do not remain wandering around in earlier parts of the level. This should, in whatever minor way, improve performance.

```
2 references
[SerializeField] private float moveSpeed;
2 references
private GameObject player;
5 references
private Rigidbody2D body;

0 references
void Awake()
{
    body = GetComponent<Rigidbody2D>();
    player = GameObject.FindWithTag("Player");
}

0 references
private void FixedUpdate()
{
    if (player.transform.position.x < transform.position.x) // if left of enemy
    {
        body.linearVelocity = new Vector3(-moveSpeed, body.linearVelocity.y, 0);
    }
    else // if right of enemy
    {
        body.linearVelocity = new Vector3(moveSpeed, body.linearVelocity.y, 0);
    }
}


1 reference
public void EnemyDeath()
{
    Destroy(gameObject); // destroy self
}
```

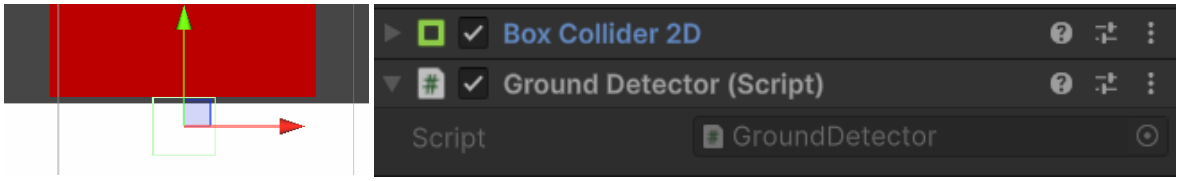
This is the truly basic framework for enemy movement. The enemy will check which direction the player is from it, and will move accordingly. In addition, the EnemyDeath function simply destroys the GameObject, but could be used for more later on in development.

```
2 references
void setMoveSpeedSign(bool positive)
{
    if (positive)
    {
        if (moveSpeed < 0)
            moveSpeed = -moveSpeed;
    }
    else
    {
        if (moveSpeed > 0)
            moveSpeed = -moveSpeed;
    }
}
```

The above function is to ensure that moveSpeed can be used to determine the direction of travel – if it is positive, the enemy is moving right, and vice versa. This will be useful in determining what to do when the enemy is on the edge of a platform.

```
private void FixedUpdate()
{
    float distance = Vector3.Distance(player.transform.position, transform.position);
    if (distance < chaseDistance)
    {
        if (player.transform.position.x < transform.position.x && transform.position.x > moveCapLeft) // if play
        {
            setMoveSpeedSign(false); // make moveSpeed negative
            body.linearVelocity = new Vector3(moveSpeed, body.linearVelocity.y, 0);
        }
        else if (player.transform.position.x > transform.position.x && transform.position.x < moveCapRight) // i
        {
            setMoveSpeedSign(true); // make moveSpeed positive
            body.linearVelocity = new Vector3(moveSpeed, body.linearVelocity.y, 0);
        }
    }
}
```

This updated version of the movement takes into account movement caps, which will be placed when the enemy encounters the edge of a platform. This must be done in a separate object which has a collider below the enemy, which can identify when the platform ends.



The Ground Detector object is given a requisite script and a box collider with IsTrigger enabled that lies below the object. The collider is so small because it only responds as having exited the ground when it is fully out, so it will respond when the collider itself is over the edge but the enemy isn't quite, so it doesn't fall.

```
1 reference
public void reachedEdge()
{
    if (moveSpeed < 0) // if moving left
    {
        moveCapLeft = transform.position.x + 0.5f; // set cap to here (little offset so doesn't hang so much)
        body.linearVelocityX = 0; // stop moving left
    }
    else
    {
        moveCapRight = transform.position.x - 0.5f; // set cap to here (little offset so doesn't hang so much)
        body.linearVelocityX = 0; // stop moving right
    }
}
```

This is the function that the groundDetector will need to call. By setting the caps with a small offset, the next time the enemy returns to them (if it does) it will overhang the edge slightly less.

```
private GameObject parentEnemy;

0 references
void OnTriggerExit2D(Collider2D collision)
{
    if (collision.CompareTag("Ground"))
    {
        Debug.Log("edge/fall");
        parentEnemy.GetComponent<Enemy>().reachedEdge(); // call reachedEdge() in enemy
    }
}

0 references
void Awake()
{
    if (transform.parent != null) // if there is a parent enemy
        parentEnemy = transform.parent.gameObject; // get enemy that this is acting for
}
```

This is the entire functionality of the ground detector. By returning when it exits the ground, it can set the caps at the correct places – the section in Awake means that the parent Enemy has its reference selected immediately, so that the detector knows which enemy should set its caps.

```
if (transform.position.x < moveCapLeft) // if beyond cap
{
    setMoveSpeedSign(true); // make moveSpeed positive
    body.linearVelocity = new Vector3(moveSpeed, body.linearVelocity.y, 0); // return to within cap
}
if (transform.position.x > moveCapRight) // if beyond cap
{
    setMoveSpeedSign(false); // make moveSpeed negative
    body.linearVelocity = new Vector3(moveSpeed, body.linearVelocity.y, 0); // return to within cap
}
```

I added this to the movement of the enemies. This ensures that they move back from hanging over the edge and can push back against another enemy pushing towards them. This does create a visual bumping and shifting on the edge that I don't love, but the function is more important and so I will not spend more time trying to resolve it. This may be addressed in a later stage, but likely not (as it does not have much of an impact.

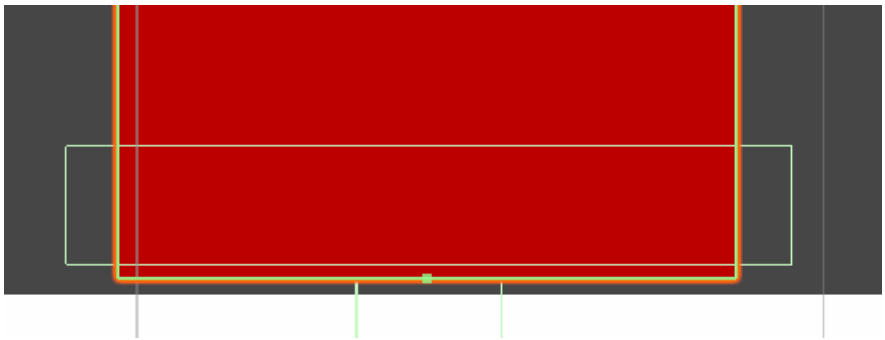
```
private void spawnEnemy()
{
    enemySpawned = true;
    spawnedEnemy = Instantiate(enemy);
    spawnedEnemy.transform.position = spawnerPosition;
    spawnedEnemy.GetComponent<Enemy>().spawnpoint = spawnerPosition; // set spawnpoint to here
}
```

```
else // player is out of range
{
    if (transform.position.x < spawnpoint.x - 3) // if left of spawnpoint
    {
        setMoveSpeedSign(true);
        body.linearVelocity = new Vector3(moveSpeed, body.linearVelocity.y, 0);
    }
    else if (transform.position.x > spawnpoint.x + 3) // if right of spanpoint
    {
        setMoveSpeedSign(false);
        body.linearVelocity = new Vector3(moveSpeed, body.linearVelocity.y, 0);
    }
    else // around spawnpoint
    {
        body.linearVelocity = new Vector3(0, body.linearVelocity.y, 0); // stamd still
    }
}
```

This piece of movement code makes it so that the enemy will return to the general area of its spawn point when the player moves out of range. This ensures that the player doesn't get blocked off from a platform (and can retreat to try again). The checkpoint is a public value which is set when the enemy is created.

```
0 references
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag(GROUND_TAG))
    {
        isGrounded = true;
        doubleJump = true;
    }
    if (collision.gameObject.CompareTag(HAZARD_TAG) && isAlive || collision.gameObject.CompareTag(ENEMY_TAG) && isAlive)
    {
        isAlive = false;
        StartCoroutine(playerDeath());
    }
}
```

This kills the player when it touches an enemy. This may be slightly edited in the Visual Retouch stage (Stage 12), if I decide to add an attack animation. This might mean that the player does not die solely on contact, but on contact with the attack.



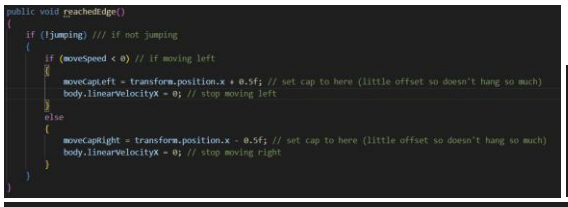
The next step is adding the ability to traverse small ledges, which may show up in level design. This is low priority, but shares many features with the GroundDetector function, so it should be relatively simple. The object has an IsTrigger collider which protrudes to get when a ledge is encountered, and a script which will be near-identical to that of groundDetector. This is slightly offset from the ground, so that it doesn't trigger while the enemy is on the ground.

```
public class JumpDetector : MonoBehaviour
{
    2 references
    private GameObject parentEnemy;

    0 references
    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Ground"))
        {
            parentEnemy.GetComponent<Enemy>().jumpLedge(); // call jumpLedge() in enemy
        }
    }

    0 references
    void Awake()
    {
        if (transform.parent != null) // if there is a parent enemy
            parentEnemy = transform.parent.gameObject; // get enemy that this is acting for
    }
}
```

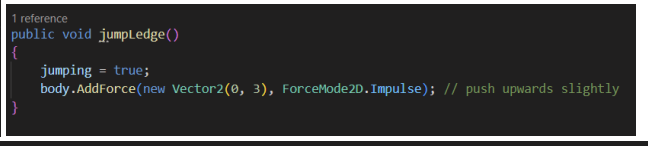
```
1 reference
public void jumpLedge()
{
    body.AddForce(new Vector2(0, 1), ForceMode2D.Impulse); // push upwards slightly
}
```

The only difference is that it calls jumpLedge() instead, and applies on entry. This is because the enemy needs to jump when it encounters a ledge, not when it leaves it. All jumpLedge() does is apply a small force upwards, to propel the enemy over the ledge.

Upon running, I realised that when the enemy jumped the groundDetector was being triggered and a cap was being set. I extended the groundDetector's collider downwards, which also means it should be able to go down small lips without stopping. When this didn't work, I tried to create some exclusions within the code – i.e. that caps cannot be set while jumping.



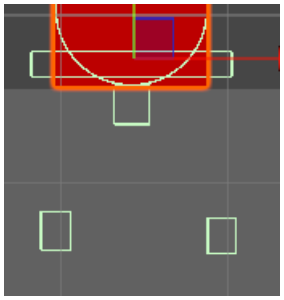
```
public void reachedEdge()
{
    if (!jumping) /// if not jumping
    {
        if (moveSpeed < 0) // if moving left
        {
            moveCapLeft = transform.position.x + 0.5f; // set cap to here (little offset so doesn't hang so much)
            body.linearVelocityX = 0; // stop moving left
        }
        else
        {
            moveCapRight = transform.position.x - 0.5f; // set cap to here (little offset so doesn't hang so much)
            body.linearVelocityX = 0; // stop moving right
        }
    }
}
```



```
1 reference
public void jumpLedge()
{
    jumping = true;
    body.AddForce(new Vector2(0, 3), ForceMode2D.Impulse); // push upwards slightly
}
```

```
0 references
void OnTriggerEnter2D(Collider2D collision) // when landing on ground
{
    if (collision.CompareTag("Ground"))
    {
        parentEnemy.GetComponent<Enemy>().jumping = false; // set jumping to false
    }
}
```

While this worked for jumping up the ledge, the enemy still could not descend.



The Lower Ledge Detectors lie on a lower offset than the GroundDetector. They are meant to detect when there is a ledge at their level that the enemy could jump down onto. Thus, they need to be able to see when there is a ledge and where there is not one, and communicate that to the overall enemy script. Each have a distinct script (with the only difference being whether it sets lowerLedgeLeft or lowerLedgeRight in their scripts) and an IsTrigger collider.

```
void Awake()
{
    if (transform.parent != null) // if there is a parent enemy
        parentEnemy = transform.parent.gameObject; // get enemy that this is acting for
    parentEnemy.GetComponent<Enemy>().ledgeBelowLeft = true; // set true initally as it will spawn on a platform
}
```

It gets the parent object, which will be the enemy it is attached to, and then sets the ledge to true as it will spawn on a platform. For brevity, I will only include the code for the left detector – the only difference is the Boolean value it changes in the parent enemy.

```
4 references
private GameObject parentEnemy;
3 references
private bool ledge;

0 references
void OnTriggerExit2D(Collider2D collision) // when exiting platform (no ledge)
{
    if (collision.CompareTag("Ground"))
    {
        ledge = false; // there is no ledge below
    }
}

0 references
void OnTriggerStay2D(Collider2D collision) // when in platform
{
    if (collision.CompareTag("Ground"))
    {
        ledge = true; // there is a ledge below
    }
}
```

A pair of collision checkers determine whether or not there is a platform there – originally I planned to have them the other way round (stay first, exit second), but I realised that if the platform is made of two different objects (e.g. a ledge atop a larger platform) then the exit of the ledge platform will be recorded, which will contradict the fact that it is still in a platform overall.

```
0 references
void Update()
{
    if (ledge)
        parentEnemy.GetComponent<Enemy>().ledgeBelowLeft = true; // there is a ledge below
    else
        parentEnemy.GetComponent<Enemy>().ledgeBelowLeft = false; // there is no ledge below
}
```

This just checks if there is a ledge each frame and puts the result into the Enemy script, which can then handle the results. Both ‘LedgeBelow’ attributes are public to make access simpler, because security concerns aren’t high and it is not an attribute that really necessitates protection.

```
public void reachedEdge()
{
    if (!jumping) /// if not jumping and no ledge below
    {
        if (moveSpeed < 0 && !ledgeBelowLeft) // if moving left & no ledge on left
        {
            moveCapLeft = transform.position.x + 0.5f; // set cap to here (little offset so doesn't hang so much)
            body.linearVelocityX = 0; // stop moving left
            Debug.Log("cap left");
        }
        else if (moveSpeed > 0 && !ledgeBelowRight) // if moving right & no ledge on right
        {
            moveCapRight = transform.position.x - 0.5f; // set cap to here (little offset so doesn't hang so much)
            body.linearVelocityX = 0; // stop moving right
            Debug.Log("cap right");
        }
    }
}
```

The updated reachedEdge function just takes into account if there is a ledge below in the direction it is moving. If there is a ledge, it does not set a cap (as it can drop safely). This concludes the traversal section of enemy development, as it can now go up and down ledges and lips in the terrain, as well as not walking off platforms and returning to their spawn after the player leaves their range.

```
0 references
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Cannonball")) // if hit by a cannonball
        EnemyDeath(); // die
}
```

```
public class Cannonball : MonoBehaviour
{
    0 references
    void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Ground") || collision.CompareTag("Hazard") || collision.CompareTag("Enemy"))
            Destroy(gameObject);
    }
}
```

This simply kills the enemy if hit by a cannonball, as well as making sure the cannonball also disappears.

```
0 references
void OnTriggerExit2D(Collider2D collision)
{
    if (collision.CompareTag("Ground") || collision.CompareTag("Hazard") || collision.CompareTag("Enemy"))
        Destroy(gameObject);
    if (collision.CompareTag("Enemy"))
        collision.GetComponent<Enemy>().EnemyDeath();
}
```

Upon running, I realised that the cannonball was deleting but the enemy was not. This is because the cannonball is an IsTrigger, so the enemy cannot register it as a collision. I changed this to use the public method EnemyDeath(), which is also called by the spawner, to kill the enemy it impacts alongside deleting itself.

Testing

Test No.	Description	Type	Data	Expected Result	Actual Result
1	Enemy sprite displayed	Valid	Program run	Enemy sprite visible	Stage 4.mp4 00:03
2	Spawn point placed	Valid	Object placed on the scene view	Spawn point tag visible in scene view	Stage 4.mp4 00:00

3a	Enemy spawns when player close	Valid	Player gets within 10 units of the spawnpoint	Enemy visible in game	Stage 4.mp4 00:03
3b	Enemy doesn't spawn when player far	Invalid	Player remains greater than 10 units away	Enemy is not visible in scene view	Stage 4.mp4 00:00
4a	Enemy moves towards player	Valid	Player moves within 7 units of the enemy	Enemy moves towards player	Stage 4.mp4 00:04
4b	Enemy can climb ledge	Valid	Player stands close to enemy on higher ledge	Enemy moves up the ledge and continues onwards	Stage 4.mp4 00:24
4c	Enemy can drop down a small ledge	Valid	Player stands close to enemy on a lower ledge	Enemy moves down the ledge and continues onwards	Stage 4.mp4 00:27
4d	Enemy does not dismount platform	Invalid	Player stands on other platform (gap between)	Enemy stands on edge of platform	Stage 4.mp4 00:31
5	Enemy returns towards spawnpoint	Valid	Player remains greater than 7 units away	Enemy returns to spawnpoint in scene view	Stage 4.mp4 00:08
6	Enemies despawn	Valid	Player moves greater than 25 units away	Enemy is not visible in scene view	Stage 4.mp4 00:15
7a	Enemies despawn when player dies	Valid	Enemy spawned, player dies	Enemy is not visible in scene view	Stage 4.mp4 00:38
7b	Enemies can respawn after player dies	Valid	Enemy killed, player dies, player moves within 7 units	Enemy is visible	Stage 4.mp4 00:53
8	Enemies kill player	Valid	Player touches enemy	Death screen occurs	Stage 4.mp4 00:36
9a	Cannonballs kill enemies	Valid	Cannonball touches enemy	Enemy disappears	Stage 4.mp4 00:44
9b	Enemies destroy cannonballs	Valid	Cannonball touches enemy	Cannonball disappears	Stage 4.mp4 00:44

Evaluation

My aims for this stage were more than for the last three, but I think they were well-met;

- display enemy sprite
- create enemy spawn point
- spawn enemies when the player gets close enough5
- enemy movement (being able to get up small hills and ledges)
- enemies go towards the player if the environment is traversable
- enemies follow for a certain distance from spawn points
- enemies return to spawn points when player leaves range
- enemies despawn when player gets far enough away
- enemies respawn when player dies
- enemies attack player at close range
- enemies are destroyed by cannonballs (and despawn cannonballs when hit)

I believe that these objectives were met well enough to create a functional game. I believe I could, as a desirable feature, make the enemy killing the player more like an attack than an on-contact death, but it is not integral and the current functionality is satisfactory. I am very happy with the movement of the enemy, which was difficult to implement in the ways that I wanted to, and I think it all runs smoothly and sufficiently.

Stage 5 – Title Screen Buttons

Because the first four stages make up the majority of the necessary gameplay functions, the GUI must now be designed. This stage involves making a mock-up of the GUI and adding in the functionality of the title screen. It should be able to transfer to both of its child screens (i.e. the New Game and Load Game screens) and quit the game entirely. In addition, the user should be able to navigate the screen with the key-binds shown.

Functionality:

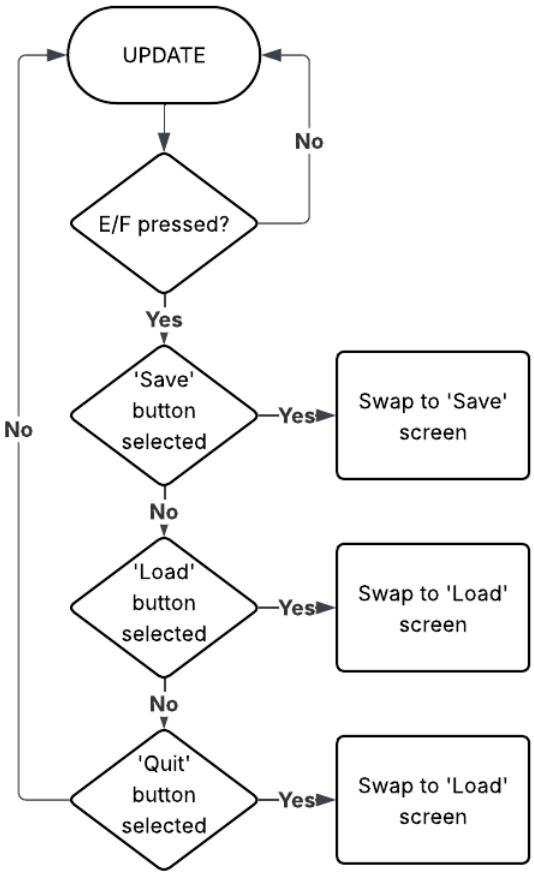
- initial GUI design added
- ‘button’ sprites (with unselected and selected variants)
- menu can be navigated with AD and Left/Right Arrows
- buttons can be selected with F and E
- transfers between requisite screens
- game can be quit

Design

Data Dictionary

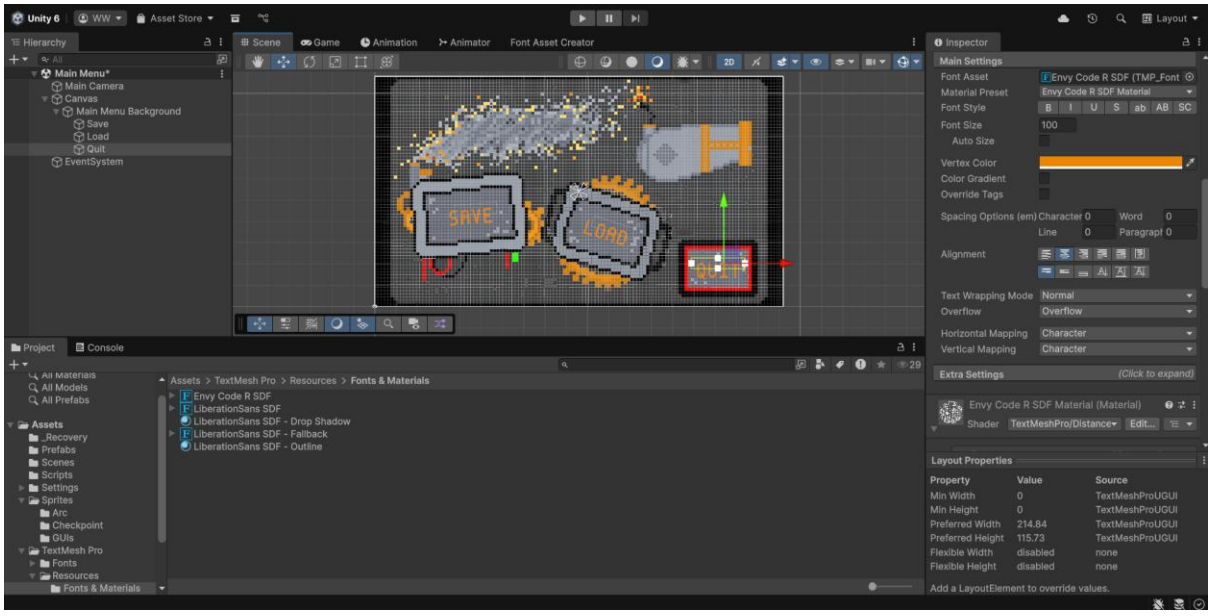
NAME	TYPE	Class	Parameters	DESCRIPTION
Awake()	void	GameManager		Executes at start of game
OnEnable()	void	GameManager		Subscribes to sceneLoaded event
OnDisable()	void	GameManager		Unsubscribes from sceneLoaded event
OnLevelFinishedLoading()	void	GameManager		Executes when a scene loads
instance	GameManager	GameManager		Holds the static instance of the manager
Update()	void	MainMenu		Executes every frame
Select()	void	MainMenu		Deals with selecting the options
Navigate()	void	MainMenu		Deals with navigation between buttons
SaveButton NewGameButton	GameObject	MainMenu		Holds reference to the Save button – renamed to New Game to better reflect purpose and GUI design
LoadButton	GameObject	MainMenu		Holds reference to the Load button
QuitButton	GameObject	MainMenu		Holds reference to the Quit button
selectedButton	int	MainMenu		Records the ‘index’ of the selected button, to render their sprites and execute the correct function
buffer	Int	MainMenu		Creates a ‘lag’ in swapping between buttons to make navigation easier

Flowchart



^within MainMenu script

Development

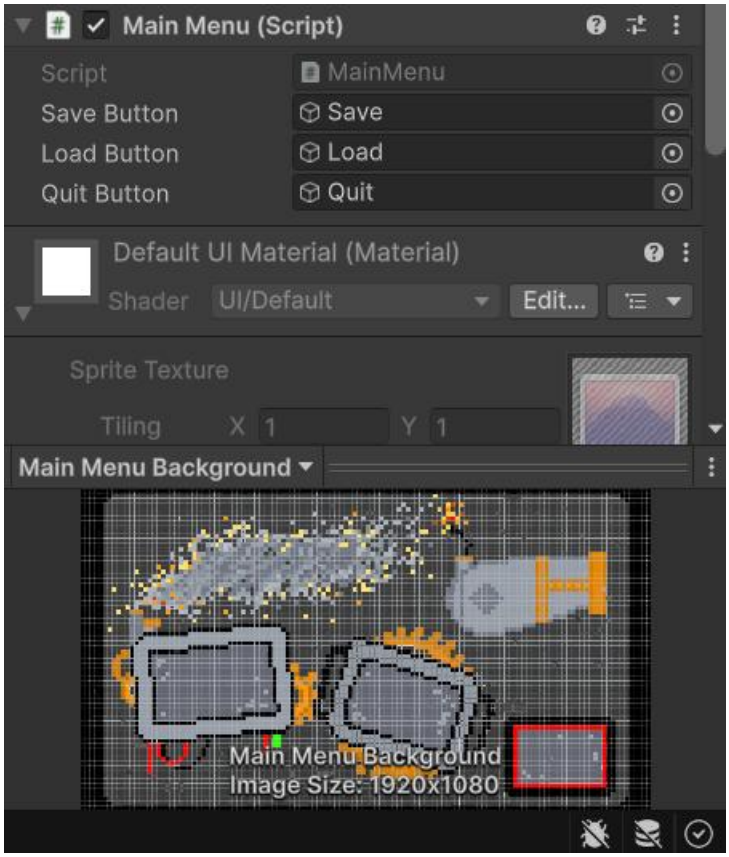


The first part of this stage is creating the UI elements involved. I created the background and aligned it to the canvas, and put in all the text where they should be. I am happy with the background as-is, but plan to

update the text to a more pixelated style in a later stage to better match with the aesthetic of the background. I also created the MainMenu script, which will control the operations of the main menu.

```
0 references
public class MainMenu : MonoBehaviour
{
    0 references
    [SerializeField] GameObject SaveButton; // index 0
    0 references
    [SerializeField] GameObject LoadButton; // index 1
    0 references
    [SerializeField] GameObject QuitButton; // index 2
    0 references
    private int selectedButton = 0; // which button is currently selected

    0 references
    void Update()
    {
        Select();
        Navigate();
    }

    1 reference
    void Select()
    {

    }

    1 reference
    void Navigate()
    {

    }
}
```

This is the framework for the MainMenu script. The selectedButton attribute will control the majority of the functions, with the references to each being used to change them visually (to show which is selected).



I added the script to the background, and assigned the objects to their requisite attributes. This allows the script to manipulate the properties of these objects.

```
void Navigate()
{
    float navigate = Input.GetAxisRaw("Horizontal");
    if (navigate < 0)
    {
        selectedButton -= 1;
        if (selectedButton < 0) // if goes out of range
            selectedButton = 2; // resets as highest value
    }
    else if (navigate > 0)
    {
        selectedButton += 1;
        if (selectedButton > 2) // if goes out of range
            selectedButton = 0; // resets as lowest value
    }


    if (selectedButton == 0)
    {
        Debug.Log("Save");
    }
    else if (selectedButton == 1)
    {
        Debug.Log("Load");
    }
    else if (selectedButton == 2)
    {
        Debug.Log("Quit");
    }
    else
    {
        Debug.Log("selectedButton out of range");
    }
}
```

Navigating the menu is basically handled by this function. Where the console commands are returned would be a visual indication of selection; the text would go bold. However, this loops far too quickly to easily control – I would like the user to be able to hold the button down to swap, so using getButtonDown would not be appropriate, but at the moment it is switching far too fast for the user to easily select. I plan to add an extra coroutine to control this, to add a short time delay to switching. This also means that I need to add Awake(), in which the coroutine is called.

```
1 reference
private IEnumerator buttonSwitch()
{
    while (true) // loop infinitely (like update but slower)
    {
        yield return new WaitForSeconds(0.1f);
        float navigate = Input.GetAxisRaw("Horizontal");
        if (navigate < 0)
        {
            selectedButton -= 1;
            if (selectedButton < 0) // if goes out of range
                selectedButton = 2; // resets as highest value
        }
        else if (navigate > 0)
        {
            selectedButton += 1;
            if (selectedButton > 2) // if goes out of range
                selectedButton = 0; // resets as lowest value
        }
    }
}
```

This function feels a little smoother and easier to navigate, but occasionally stutters and can still be tricky to deal with. I realised that using a coroutine wouldn't work because there would be no response after pressing a button if it was within the 'off' period of the coroutine, which would feel clunky in other moments.

```
float navigate = Input.GetAxisRaw("Horizontal");
if (navigate < 0 && buffer == 0)
{
    selectedButton -= 1;
    if (selectedButton < 0) // if goes out of range
        selectedButton = 2; // resets as highest value
    buffer = 60;
}
else if (navigate > 0 && buffer == 0)
{
    selectedButton += 1;
    if (selectedButton > 2) // if goes out of range
        selectedButton = 0; // resets as lowest value
    buffer = 60;
}
else if (buffer > 0)
{
    buffer--;
}
if (navigate == 0)
{
    buffer -= 10;
    if (buffer < 0)
        buffer = 0;
}
```

I decided to return to the Update() function, adding an integer attribute called buffer which will create a pause in when the system can switch between buttons, which will also allow the user to press the buttons and always get a response, without it moving too quickly or skipping over buttons. I decided to reduce the buffer value to 20 instead of 60 to make the movement feel more natural. This creates a quick movement through the buttons on the screen, without it being too quick or unwieldy. This was clearly a better option to the easier-to-implement but stuttering coroutine I tried first.

```
if (selectedButton == 0)
{
    SaveButton.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Bold; // make bold
    LoadButton.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Normal; // make normal
    QuitButton.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Normal; // make normal
}
else if (selectedButton == 1)
{
    SaveButton.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Normal; // make normal
    LoadButton.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Bold; // make bold
    QuitButton.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Normal; // make normal
}
else if (selectedButton == 2)
{
    SaveButton.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Normal; // make normal
    LoadButton.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Normal; // make normal
    QuitButton.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Bold; // make bold
}
else
{
    Debug.Log("selectedButton out of range"); // show that some error has occurred
}
```

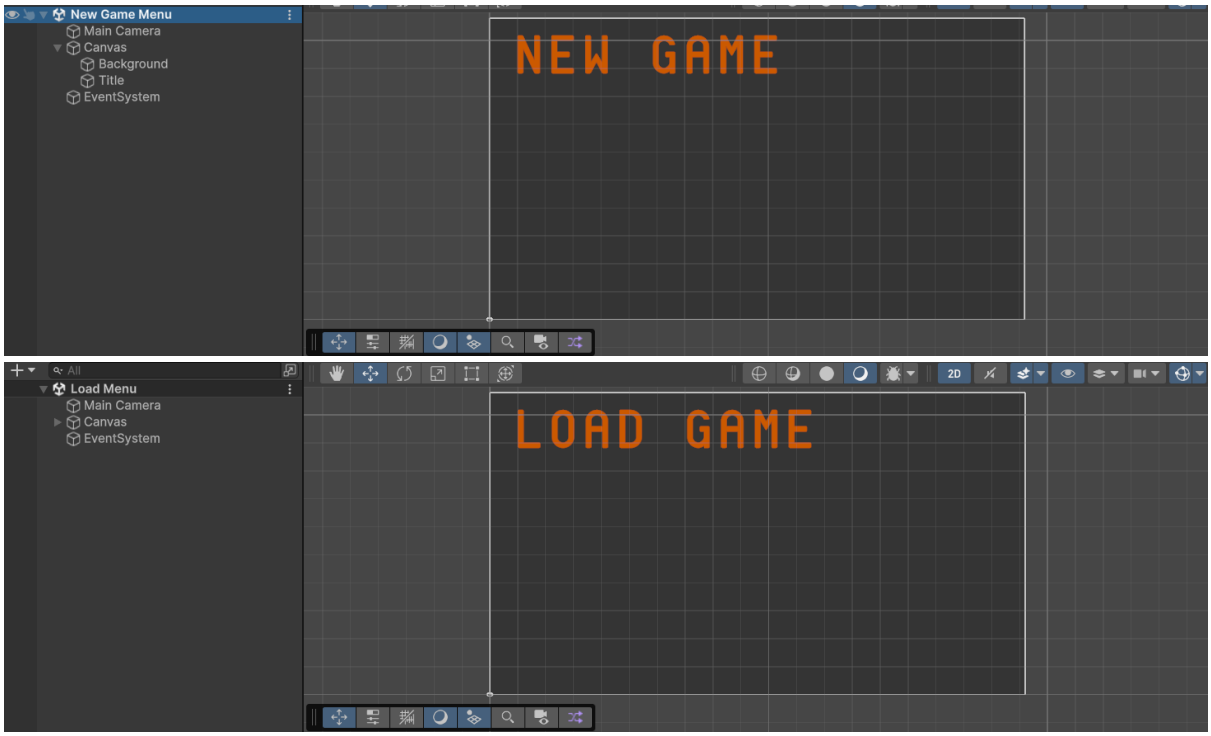
This is the section of code that allows the buttons to go bold (to indicate selection). Each simply ensures that the correct button is bold, while the others are normal. This is likely to change in the visual retouch stage, as I change what the buttons look like and perhaps how the selection visibility looks, as it not overtly clear at the moment.

```
if (Input.GetKeyDown(KeyCode.E) || Input.GetKeyDown(KeyCode.F))
{
    if (selectedButton == 0) // save
    {

    }
    else if (selectedButton == 1) // load
    {

    }
    else if (selectedButton == 2) // quit
    {

    }
}
```

```
1 reference
void Select()
{
    if (Input.GetKeyDown(KeyCode.E) || Input.GetKeyDown(KeyCode.F))
    {
        if (selectedButton == 0) // save
        {
            Debug.Log("Save Scene");
            SceneManager.LoadScene("Save Menu");
        }
        else if (selectedButton == 1) // load
        {
            Debug.Log("Load Scene");
            SceneManager.LoadScene("Load Menu");
        }
        else if (selectedButton == 2) // quit
        {
            Debug.Log("Quit");
            Application.Quit();
        }
    }
```

This will get when the Interact key is pressed and perform a process determined by the currently selected button. The console outputs are to inform me whether the scene should be changing or if the application would quit – that command doesn't work in the play mode of the Unity Editor, and the other scenes do not yet exist.

I remembered that it was not in fact a 'save' button, but a new game one, and so I renamed everything appropriately.



Next, I created the scenes themselves, so that the main menu had something to transfer into. This marks the end of Stage 5, as all functionality for the above scenes will be dealt with in Stage 6.

Testing

Test No.	Description	Type	Data	Expected Result	Actual Result
1a	GUI shown on screen	Valid	Run program on Title Screen scene	Title Screen GUI visible	Stage 5 (1).mp4 00:00

1b	Button sprites visible	Valid	Run program on Title Screen scene	All text for the 'buttons' visible, with New Game in bold	Stage 5 (1).mp4 00:00
2a	'Right Arrow' works	Valid	'Right Arrow' pressed, New Game bold	Next button (load) becomes bold	Stage 5 (1).mp4 00:04
2b	'D' works	Valid	'D' pressed, Load Game bold	Next button (quit) becomes bold	Stage 5 (1).mp4 00:05
2c	'Left Arrow' works	Valid	'Left Arrow' pressed, Quit Game bold	Previous button (load) becomes bold	Stage 5 (1).mp4 00:02
2d	'A' works	Valid	'A' pressed, Load Game bold	Previous button (save) becomes bold	Stage 5 (1).mp4 00:03
3a	Hold left is not too fast	Invalid	'A' held	Buttons switch with decent pace	Stage 5 (1).mp4 00:04
3b	Hold right is not too fast	Invalid	'D' held	Buttons switch with decent pace	Stage 5 (1).mp4 00:07
4a	New Game button works	Valid	New Game bold, 'E' pressed	Transfers to New Game screen	Stage 5 (1).mp4 00:21
4b	Load Game button works	Valid	Load bold, 'F' pressed	Transfers to Load Game screen	Stage 5 (2).mp4 00:05
4c	Quit button works	Valid	Quit bold, 'E' pressed	Console reads 'Quit'	Stage 5 (3).mp4 00:01 (console)

Evaluation

The functionality I wanted to implement in Stage 5 was:

- initial GUI design added
- 'button' sprites (with unselected and selected variants)
- menu can be navigated with AD and Left/Right Arrows
- buttons can be selected with F and E
- transfers between requisite screens
- game can be quit

I think all these objectives were well met, and the GUI is largely similar to my end position, relying on the GUI designs I created in the Design phase. Some of the code may be altered depending on sprites and the like in the Visual Retouch stage (12).

Stage 6 – Saving/Loading

To give any true functionality (beyond navigation) to the title screen, the game must be able to load saves or create a new save file. This stage concerns the creation and accessing of these saves, the format in which I will save data, how the saves will be presented on the save screen, and ensuring that checkpoints can save data. I also need to balance the four save slots and make sure that they don't overwrite each other or cause conflicts. By adding this here, it effectively finalises the title screen (beyond any aesthetic retouches in Stage 12), which ensures that another essential part of the program is complete.

Functionality:

- checkpoints save required game data to the requisite file
- game data includes current checkpoint and any upgrades
- new game select can create a new save file
- new game select can overwrite an existing save file
- load game can read the requisite file to start a game from that position
- save files persist beyond the game being terminated

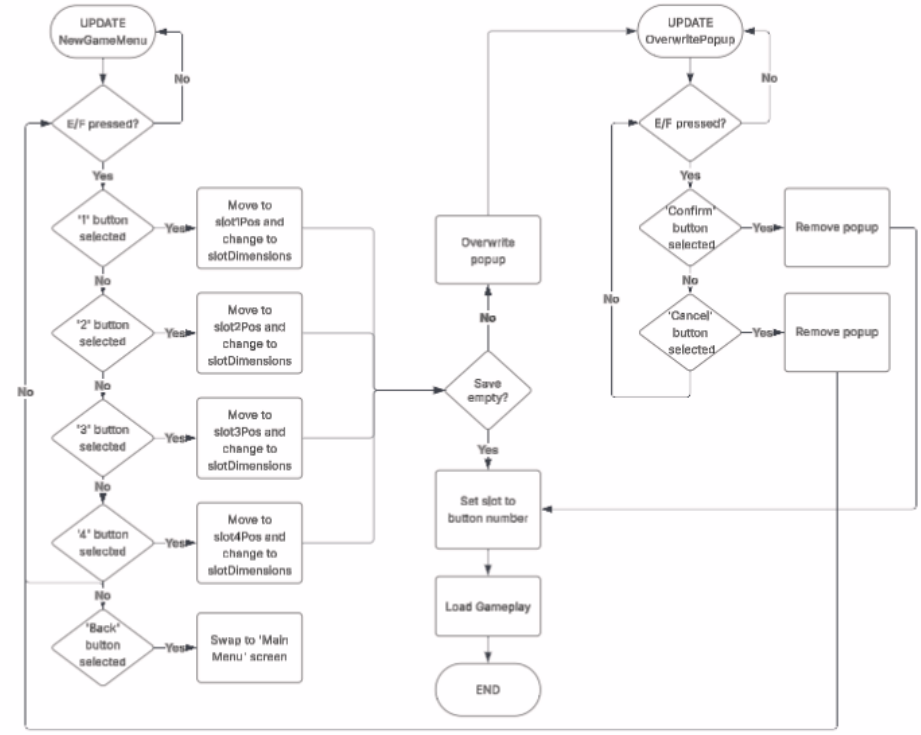
Design

Data Dictionary

NAME	TYPE	Class	Parameters	DESCRIPTION
Update()	void	NewGameMenu, SaveOverwrite, LoadGameMenu, GameManager		Executes every frame
Select()	void	NewGameMenu, SaveOverwrite, LoadGameMenu		Handles selecting the button
Navigate()	void	NewGameMenu, SaveOverwrite, LoadGameMenu		Handles moving between buttons
ShowData()	void	NewGameMenu, LoadGameMenu		Shows the required save data in slots
checkSaveEmpty()	void	NewGameMenu, LoadGameMenu	int n	Checks if a save is empty
overwriteCheck()	void	NewGameMenu, LoadGameMenu	int slot	Handles what occurs when overwriting a save with contents
boxSelector	GameObject	NewGameMenu, LoadGameMenu		Holds reference to boxSelector object
saveSlotDimensions	Vector2	NewGameMenu, LoadGameMenu		Holds dimensions of boxSelector when it is used to select a save slot
slot1Pos	Vector3	NewGameMenu, LoadGameMenu		Holds position of slot1 for box
slot2Pos	Vector3	NewGameMenu, LoadGameMenu		Holds position of slot2 for box
slot3Pos	Vector3	NewGameMenu, LoadGameMenu		Holds position of slot3 for box
slot4Pos	Vector3	NewGameMenu, LoadGameMenu		Holds position of slot4 for box
backDimensions	Vector2	NewGameMenu, LoadGameMenu		Holds dimensions of boxSelector when it is used to select the back button
backPos	Vector3	NewGameMenu, LoadGameMenu		Holds the position of back button for box
selectedButton	int	NewGameMenu, LoadGameMenu		Holds ‘index’ of button
buffer	int	NewGameMenu, LoadGameMenu, SaveOverwrite		Value that applies some ‘lag’ to navigating buttons, to make it easier to use
overwritePopup	bool	NewGameMenu		Checks if the overwritePopup is currently active, so functions should be disabled
OverwritePopup	bool	NewGameMenu		Public version of the above
overwrite	void	SaveOverwrite		Holds reference to overwrite button
cancel	void	SaveOverwrite		Holds reference to cancel button
FadePopup()	void	SaveOverwrite, EmptySavePopup		Fades the popup (in or out)
selectBuffer	Int	SaveOverwrite		Stops the buttons from being clicked immediately after the popup appears
Fade()	void	FadeText		Fades the text in or out

objectToFade	GameObject	FadeText		Holds object for fading
fadeSpeed	int	FadeText		Speed of fading
Awake()	void	GameManager		Executes at start, holds Singleton
instance	static GameManager	GameManager		Holds static instance of GameManager class for reference
saveSlot	int	GameManager		Save slot in use in the game
SaveSlot	int	GameManager		Public of above
player	GameObject	GameManager		Reference to player
Player	GameObject	GameManager		Public of above
SaveFileName()	static string	SaveSystem		Gets/creates file name of save
Save()	static void	SaveSystem		Saves game
HandleSaveData()	static void	SaveSystem		Gets save data from different objects
Load()	static void	SaveSystem		Loads game
HandleLoadData()	static void	SaveSystem		Gives save data to different objects
readFileName()	static string	SaveSystem		Get time from save file
readFileScene()	static string	SaveSystem		Get scene from save file
Save()	void	Plater	PlayerSaveData	Saves player data
Load()	void	Player	PlayerSaveData	Loads player data
PlayerSaveData	struct	Player		Holds data that needs to be saved
findCheckpoint	bool	Player		T/F for whether a checkpoint object needs to be found to assign
popup()	void	LoadGameMenu		Shows popup when empty save selected
emptyPopup	bool	LoadGameMenu		Tracks whether the above popup is shown
saveSlot	int	EmptySavePopup		Holds save slot

Pseudocode/Flowchart



```
CLASS GameManager
    PUBLIC STATIC GameManager instance
    PRIVATE INT saveSlot
    PUBLIC INT SaveSlot (getter and setter here)
    PRIVATE GAMEOBJECT player
    PUBLIC GAMEOBJECT Player (getter and setter here)

    PROCEDURE Awake()
        IF instance == NULL THEN
            intstance = THIS
            DONTDESTROYONLOAD(gameObject)
        ELSE
            DESTROY(gameObject)
        ENDIF
        player = GAMEOBJECT.FIND("Player")
    ENDPROCEDURE

    PROCEDURE Update()
        IF player == NULL THEN
            player = GAMEOBJECT.FIND("Player")
        ENDIF
    ENDPROCEDURE
ENDCLASS
```

I have not included any design for the SaveSystem class as it will be composed of resources from the tutorial mentioned below. Any additional sections that may be added will be outlined in the Development stage.

Development

I created the placeholder GUI for the New Game menu – most of the following will cross over to the Load menu as well, although I will note any places that it doesn't.

```
2 references
[SerializeField] GameObject boxSelector;
0 references
private Vector2 saveSlotDimensions = new Vector2(854, 295);
1 reference
private Vector3 slot1Pos = new Vector3(-439, 120);
0 references
private Vector3 slot2Pos = new Vector3(-439, -175, 0);
0 references
private Vector3 slot3Pos = new Vector3(505, 120, 0);
0 references
private Vector3 slot4Pos = new Vector3(505, -175, 0);
0 references
private Vector2 backDimensions = new Vector2(370, 150);
```

I created the script that would control the new game menu, hardcoding the positions of the slots and the dimensions of the box selector so that it would wrap around the save slot or back button adequately.

```
1 reference
void Navigate()
{
    float navigate = Input.GetAxisRaw("Horizontal");
    if (navigate < 0 && buffer == 0)
    {
        selectedButton -= 1;
        if (selectedButton < 0) // if goes out of range
            selectedButton = 4; // resets as highest value
        buffer = 20;
    }
    else if (navigate > 0 && buffer == 0)
    {
        selectedButton += 1;
        if (selectedButton > 4) // if goes out of range
            selectedButton = 0; // resets as lowest value
        buffer = 20;
    }
    else if (buffer > 0)
    {
        buffer--;
    }
    if (navigate == 0)
    {
        buffer -= 10;
        if (buffer < 0)
            buffer = 0;
    }
```

```
if (selectedButton == 0) //  save 1
{

}
else if (selectedButton == 1) // save 2
{

}
else if (selectedButton == 2) // save 3
{

}
else if (selectedButton == 3) // save 4
{

}
else if (selectedButton == 4) // back button
{

}
else
{
    Debug.Log("selectedButton out of range"); // show that some error has occurred
}
```

I copied over the functionality of the navigation from Main Menu, changing it to work with 5 buttons instead of three. I also changed the positions of Slot 2 and Slot 3 with each other in order to make more sense.

```
RectTransform rectTransform = boxSelector.GetComponent<RectTransform>();
if (selectedButton == 0) //  save 1
{
    rectTransform.localPosition = slot1Pos;
    rectTransform.sizeDelta = saveSlotDimensions;
}
else if (selectedButton == 1) // save 2
{
    rectTransform.localPosition = slot2Pos;
    rectTransform.sizeDelta = saveSlotDimensions;
}
else if (selectedButton == 2) // save 3
{
    rectTransform.localPosition = slot3Pos;
    rectTransform.sizeDelta = saveSlotDimensions;
}
else if (selectedButton == 3) // save 4
{
    rectTransform.localPosition = slot4Pos;
    rectTransform.sizeDelta = saveSlotDimensions;
}
else if (selectedButton == 4) // back button
{
    rectTransform.localPosition = backPos;
    rectTransform.sizeDelta = backDimensions;
}
else
{
    Debug.Log("selectedButton out of range"); // show that some error has occurred
}
```

This piece of code moves the outlining box to the correct position and sets it to the right dimensions. This is the navigation of the New Game screen completed.

```
1 reference
void Select()
{
    if (Input.GetKeyDown(KeyCode.E) || Input.GetKeyDown(KeyCode.F))
    {
        if (selectedButton == 0) // slot 1
        {

        }
        else if (selectedButton == 1) // slot 2
        {

        }
        else if (selectedButton == 2) // slot 3
        {

        }
        else if (selectedButton == 3) // slot 4
        {

        }
        else if (selectedButton == 4) // back
        {
            SceneManager.LoadScene("Main Menu"); // go to main menu
        }
    }
}
```

The select function was next to be created, with only the functionality for the back button created at the moment – the other three require some level of file management to check for empty saves and rewrite them, or to make new saves. For file management in this stage, I will be using principles from this tutorial. However, a lot of ideas will need to be determined experimentally or through extrapolation from these techniques, as my save system is slightly different to the one discussed in the video.

```
public class GameManager : MonoBehaviour
{
    3 references
    public static GameManager instance;
    2 references
    private int saveSlot;
    1 reference
    public int SaveSlot { get { return saveSlot; } set { saveSlot = value; } }

    0 references
    void Awake()
    {
        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }

}
```

```
2 references
public static string SaveFileName()
{
    int saveSlot = GameManager.instance.SaveSlot;
    string saveFile = "";
    if (saveSlot == 1)
    {
        saveFile = Application.persistentDataPath + "slot1" + ".save";
    }
    else if (saveSlot == 2)
    {
        saveFile = Application.persistentDataPath + "slot2" + ".save";
    }
    else if (saveSlot == 3)
    {
        saveFile = Application.persistentDataPath + "slot3" + ".save";
    }
    else if (saveSlot == 4)
    {
        saveFile = Application.persistentDataPath + "slot4" + ".save";
    }
    else
    {
        return null;
    }
    return saveFile;
}
```

This was the first major divergence from the code in the video. As I want four distinct save slots, there must be four files. This uses a value stored in the GameManager (that will be set at the save/load screens) to determine the name of the file that data should be saved to or loaded from.

```
4 references
bool checkSaveEmpty(string fileName)
{
    bool emptySlot = false;
    if (File.Exists(fileName))
    {
        string fileContents = File.ReadAllText(fileName);
        if (fileContents == null || fileContents == "")
        {
            emptySlot = true;
        }
    }
    else
    {
        emptySlot = true;
    }
    return emptySlot;
}
```

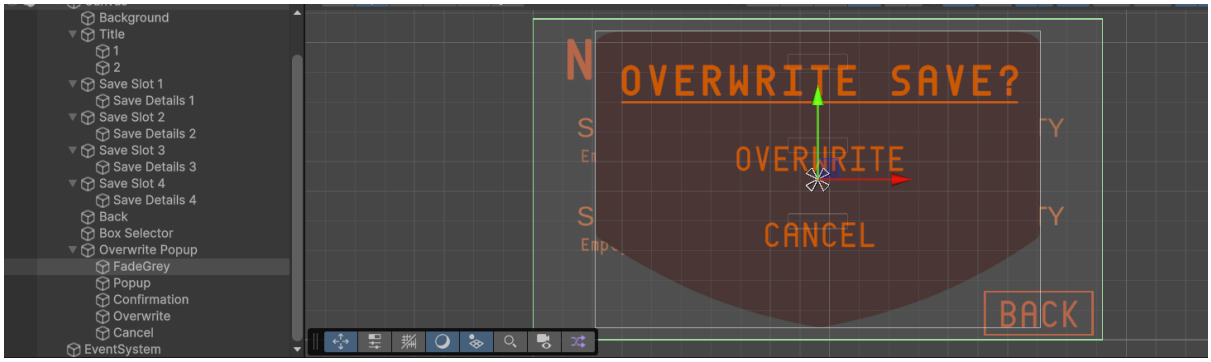
This is another method of my own devising, which should check for whether a save is empty (i.e. no such file or an empty one) and return a Boolean value.

```
1 reference
void ShowData()
{
    string fileName;
    fileName = Application.persistentDataPath + "slot1" + ".save";
    if (checkSaveEmpty(fileName))
    {
        GameObject.Find("Save Slot 1").GetComponent<TextMeshProUGUI>().text = "SLOT 1 - EMPTY"; // add data
        GameObject.Find("Save Details 1").GetComponent<TextMeshProUGUI>().text = "Empty Save"; // add data
    }
    else
    {
        GameObject.Find("Save Slot 1").GetComponent<TextMeshProUGUI>().text = "SLOT 1 - "; // add level
        GameObject.Find("Save Details 1").GetComponent<TextMeshProUGUI>().text = "Last Saved: "; // add time
    }
    fileName = Application.persistentDataPath + "slot2" + ".save";
    if (checkSaveEmpty(fileName))
    {
        GameObject.Find("Save Slot 2").GetComponent<TextMeshProUGUI>().text = "SLOT 2 - EMPTY"; // add data
        GameObject.Find("Save Details 2").GetComponent<TextMeshProUGUI>().text = "Empty Save"; // add data
    }
    else
    {
        GameObject.Find("Save Slot 2").GetComponent<TextMeshProUGUI>().text = "SLOT 2 - "; // add level
        GameObject.Find("Save Details 2").GetComponent<TextMeshProUGUI>().text = "Last Saved: "; // add time
    }
    fileName = Application.persistentDataPath + "slot3" + ".save";
    if (checkSaveEmpty(fileName))
    {
        GameObject.Find("Save Slot 3").GetComponent<TextMeshProUGUI>().text = "SLOT 3 - EMPTY"; // add data
        GameObject.Find("Save Details 3").GetComponent<TextMeshProUGUI>().text = "Empty Save"; // add data
    }
    else
```

This method, also without the help of the tutorial, should be able to (checking if the save is empty or not) set the details to a vague reflection of what it contains. By the time this stage is complete, it should show the time of save – it cannot yet say the level as they do not exist as such at the moment. The time will be dealt with when I get to using checkpoints to save data.

```
    if (selectedButton == 0) // slot 1
    {
        GameManager.instance.SaveSlot = 1;
    }
    else if (selectedButton == 1) // slot 2
    {
        GameManager.instance.SaveSlot = 2;
    }
    else if (selectedButton == 2) // slot 3
    {
        GameManager.instance.SaveSlot = 3;
    }
    else if (selectedButton == 3) // slot 4
    {
        GameManager.instance.SaveSlot = 4;
    }
    else if (selectedButton == 4) // back
    {
        SceneManager.LoadScene("Main Menu"); // go to main menu
    }
}
```

This allows the buttons to select the saveSlot the player wants to use. Before this happens, the overwrite prompt must appear (if pertinent) and the user should be able to backtrack if they wish.



I renamed some of the fadeUI attributes and methods to better represent a general purpose script, and attached that to a placeholder popup. The popup also has a slightly transparent grey background, to decrease the priority of the inaccessible GUI behind it.

```
    private bool overwritePopup;
    0 references
    public bool OverwritePopup { get { return overwritePopup; }  set { overwritePopup = value; } }

    0 references
    void Update()
    {
        if (!overwritePopup)
        {
            ShowData();
            Select();
            Navigate();
        }
    }
```

```
4 references
private void overwriteCheck()
{
    GameObject popup = GameObject.Find("Overwrite Popup");
    popup.GetComponent<FadeUI>().Fade(true);
    overwritePopup = true;
}
```

When the overwrite popup is called, it removes functionality from the screen behind it. Instead, all navigation and selection will occur on the screen in front – this will echo the code for navigating the main menu, but for only two buttons and with the vertical axis. The buffer will also be higher, because it is a more important decision and so it should be more difficult to accidentally select the wrong option. The overwritePopup variable will be accessible for other classes so that the saveOverwrite script can remove the block when it is selected.

```
if (selectedButton == 0) // slot 1
{
    string fileName = Application.persistentDataPath + "slot1" + ".save";
    if (!checkSaveEmpty(fileName))
    {
        overwriteCheck(1);
    }
    GameManager.instance.SaveSlot = 1;
}
else if (selectedButton == 1) // slot 2
{
    string fileName = Application.persistentDataPath + "slot2" + ".save";
    if (!checkSaveEmpty(fileName))
    {
        overwriteCheck(2);
    }
    GameManager.instance.SaveSlot = 2;
}
else if (selectedButton == 2) // slot 3
{
    string fileName = Application.persistentDataPath + "slot1" + ".save";
```

This checks if the requisite file exist. If it does, it begins the check, passing in the saveSlot – this is because the popup will then pass that attribute into the gameManager. If the file does not exist, the code goes ahead with no problems.

```
4 references
private void overwriteCheck(int slot)
{
    GameObject popup = GameObject.Find("Overwrite Popup");
    popup.GetComponent<FadeUI>().Fade(true);
    overwritePopup = true;
    popup.GetComponent<SaveOverwrite>().saveSlot = slot;
}
```

```
0 references
[SerializeField] GameObject overwrite;
0 references
[SerializeField] GameObject cancel;
2 references
public int saveSlot = 0;
0 references
public int selectedButton = 1; // starts on cancel
0 references
public int buffer = 0;

0 references
void Update()
{
    if (saveSlot != 0)
    {
        Select();
        Navigate();
    }
}

1 reference
void Select()
{

}

1 reference
void Navigate()
{

}
```

This piece of code allows the NewGameMenu to set the saveSlot in the overwrite popup to something greater than 0, which in turn starts the rest of the code. When an object is selected, it will reset the local attribute back to 0. Both the navigate and select methods will be very similar to those used in the MainMenu. **The selectedButton and buffer attributes are not meant to be public – this was changed a little later on when I noticed.**

```
1 reference
void Navigate()
{
    float navigate = Input.GetAxisRaw("Vertical");
    if (navigate > 0 && buffer == 0)
    {
        selectedButton -= 1;
        if (selectedButton < 0) // if goes out of range
            selectedButton = 1; // resets as highest value
        buffer = 30;
    }
    else if (navigate < 0 && buffer == 0)
    {
        selectedButton += 1;
        if (selectedButton > 1) // if goes out of range
            selectedButton = 0; // resets as lowest value
        buffer = 30;
    }
    else if (buffer > 0)
    {
        buffer--;
    }
    if (navigate == 0)
    {
        buffer -= 10;
        if (buffer < 0)
            buffer = 0;
    }
```

```
if (selectedButton == 0) // overwrite
{
    overwrite.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Bold;
    cancel.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Normal;
}
else if (selectedButton == 1) // cancel
{
    overwrite.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Normal;
    cancel.GetComponent<TextMeshProUGUI>().fontStyle = FontStyles.Bold;
}
else
{
    Debug.Log("selectedButton out of range");
}
```

```
1 reference
void Select()
{
    if (Input.GetKeyDown(KeyCode.E) || Input.GetKeyDown(KeyCode.F))
    {
        if (selectedButton == 0) // overwrite
        {
            GameManager.instance.SaveSlot = saveSlot;
            SceneManager.LoadScene("Gameplay");
        }
        else if (selectedButton == 1) // cancel
        {
            gameObject.GetComponent<FadeUI>().Fade(false);
            saveSlot = 0;
            GameObject.Find("New Game Menu").GetComponent<NewGameMenu>().OverwritePopup = false;
        }
    }
}
```

The buffer values were altered, as were the number of buttons, and the selected buttons now either load the gameplay (since it will begin from the start) or return to the previous screen (without the popup). This may need to be slightly edited after level design, as scene names change or the starting position differs.

```
3 references
public void FadePopup(bool fadeBool)
{
    foreach (Transform child in transform)
    {
        child.gameObject.GetComponent<FadeUI>().Fade(fadeBool);
    }
}
```

As the popup was not fading, I decided to add a new function which faded each child component individually. I replaced all calls to the original with calls to this. This means that I could determine that the text elements could not fade, as they do not have image components.

```
3 references
public void FadePopup(bool fadeBool)
{
    foreach (Transform child in transform)
    {
        Debug.Log(child.gameObject.name);
        try // if has  UI
        {
            StartCoroutine(child.gameObject.GetComponent<FadeUI>().Fade(fadeBool));
        }
        catch // if doesn't have UI (must have text)
        {
            StartCoroutine(child.gameObject.GetComponent<FadeText>().Fade(fadeBool));
        }
    }
}
```

```
1 reference
public IEnumerator Fade(bool fadeIn)
{
    Color objectColor = objectToFade.GetComponent<TextMeshProUGUI>().color; // get reference to colour
    float fadeAmount;
    if (fadeIn) // if fading to black
    {
        while (objectToFade.GetComponent<TextMeshProUGUI>().color.a < 1) // continue until fully opaque
        {
            fadeAmount = objectColor.a + (fadeSpeed * Time.deltaTime); // how much the thing should fade by
            objectColor = new Color(objectColor.r, objectColor.g, objectColor.b, fadeAmount); // change colour by fade amount
            objectToFade.GetComponent<TextMeshProUGUI>().color = objectColor; // set this colour to the screen
            yield return null; // stop code when completed loop
        }
    }
    else // if fading from black
    {
        while (objectToFade.GetComponent<TextMeshProUGUI>().color.a > 0) // continue until fully transparent
        {
            fadeAmount = objectColor.a - (fadeSpeed * Time.deltaTime); // how much the thing should fade by
            objectColor = new Color(objectColor.r, objectColor.g, objectColor.b, fadeAmount); // change colour by fade amount
            objectToFade.GetComponent<TextMeshProUGUI>().color = objectColor; // set this colour to the screen
            yield return null; // stop code when completed loop
        }
    }
}
```

I used a copied over version of the FadeUI script and changed it to work for TextMeshProUI, and added a try-catch function that will call the FadeUI if it exists – if it doesn't, it will call the FadeText instead.

```
if (selectedButton == 0) // slot 1
{
    string fileName = Application.persistentDataPath + "slot1" + ".save";
    if (!checkSaveEmpty(fileName))
    {
        overwriteCheck(1);
    }
    GameManager.instance.SaveSlot = 1;
    SceneManager.LoadScene("Gameplay");
```

I then added to every save slot button on the New Game menu that it would go to the gameplay as long as it was empty. The next step of this stage is to code the saving itself, and what is saved, by the checkpoints.

```
if (Input.GetKeyDown(KeyCode.E) || Input.GetKeyDown(KeyCode.F)) // if press E or F
{
    activateCheckpoint();
}
```

```
1 reference
public void activateCheckpoint()
{
    StopCoroutine(popup(true)); // stop any coroutine showing the popup
    StartCoroutine(popup(false)); // start hiding the popup
    popup(false); // remove popUp
    GameObject.FindWithTag("Player").GetComponent<Player>().setRespawnPosition(checkpointPosition); // set position to new posi
    activeCheckpoint = true;
    animator.SetBool("activeCheckpoint", true);
}
```

I moved the part of the checkpoint that activates it to a separate public variable, so that it can be called when loaded. Otherwise, the player would appear in the correct position but the checkpoint would not be selected as the respawn point, which does not make much sense.

```
1 reference
public void setActiveCheckpoint(GameObject checkpoint)
{
    activeCheckpoint = checkpoint; // get object
    respawnPosition = checkpoint.GetComponent<Checkpoint>().getPosition(); // get respawn position
}
```

I then changed the setRespawnPosition to setActiveCheckpoint, so the player could have a reference to the checkpoint it will respawn at (for saving and loading at it). I did not change that it would also get respawnPosition, as that is used in a number of other functions and I did not want to inadvertently break something.

```
1 reference
public void activateCheckpoint()
{
    StopCoroutine(popup(true)); // stop any coroutine showing the popup
    StartCoroutine(popup(false)); // start hiding the popup
    popup(false); // remove popUp
    GameObject.FindWithTag("Player").GetComponent<Player>().setActiveCheckpoint(gameObject); // set active checkpoint to this o
    activeCheckpoint = true;
    animator.SetBool("activeCheckpoint", true);
}

1 reference
public Vector3 getPosition()
{
    return checkpointPosition;
}
```

This just shows how the checkpoint sets itself into the player, and the new getPosition method.

```
    #region saveAndLoad

    0 references
    public void Save(PlayerSaveData data)
    {
        data.checkpoint = activeCheckpoint; // save the value of the activeCheckpoint
    }

    0 references
    public void Load(PlayerSaveData data)
    {
        activeCheckpoint = data.checkpoint; // set activeCheckpoint to the value that has been loaded
        activeCheckpoint.GetComponent<Checkpoint>().activateCheckpoint(); // activate it
        respawnPosition = activeCheckpoint.GetComponent<Checkpoint>().getPosition(); // get respawn position
    }

    #endregion
}

[System.Serializable]
2 references
public struct PlayerSaveData
{
    2 references
    public GameObject checkpoint;
}
```

I set up the region and struct in player as shown in the video. I decided to save in the player itself (using the checkpoint) because it will likely hold data I may need to save later, like upgrades or similar. In addition, there is only ever one player, so no duplicate saves will occur.

```
[System.Serializable]
3 references
public struct SaveData
{
    1 reference
    public PlayerSaveData PlayerData;
}
```

```
1 reference
private static void HandleSaveData()
{
    GameManager.instance.Player.GetComponent<Player>().Save(_saveData.PlayerData);
}
```

```
1 reference
private static void HandleLoadData()
{
    GameManager.instance.Player.GetComponent<Player>().Load(_saveData.PlayerData); // call load in Player
}
```

I put in the playerSavaData to the SaveSystem wherever it was required (according to the tutorial).

```
2 references
public void activateCheckpoint()
{
    StopCoroutine(popup(true)); // stop any coroutine showing the popup
    StartCoroutine(popup(false)); // start hiding the popup
    popup(false); // remove popUp
    GameObject.FindWithTag("Player").GetComponent<Player>().setActiveCheckpoint(
    activeCheckpoint = true;
    animator.SetBool("activeCheckpoint", true);
    SaveSystem.Save(); // save!
}
```

I then ensured that when the checkpoint was activated it would save to file.

```
1 reference
public void Load(PlayerSaveData data)
{
    activeCheckpoint = data.checkpoint; // set activeCheckpoint to the value that has been loaded
    activeCheckpoint.GetComponent<Checkpoint>().activateCheckpoint(); // activate it
    respawnPosition = activeCheckpoint.GetComponent<Checkpoint>().getPosition(); // get respawn position
    transform.position = respawnPosition; // move to that position
}

#endregion
```

I added a line to the Load in player, where it would move to the position of its respawn.

```
[09:52:08] NullReferenceException: Object reference not set to an instance of an object
SaveSystem.SaveFileName () (at Assets/Scripts/SaveSystem.cs:17)
```

This error kept appearing, seeming to be something to do with the GameManager.instance. I haven't figured out a way to perfectly fix this error – it does not appear when first running the game, only after editing something in the script and forcing the editor to reload. In this way, it does not feel overly important, as it should not affect the game itself.

```
if (activeCheckpoint != null)
{
    activeCheckpoint.GetComponent<Checkpoint>().activateCheckpoint(); // activate it
    respawnPosition = activeCheckpoint.GetComponent<Checkpoint>().getPosition(); // get respawn position
    transform.position = respawnPosition; // move to that position
}
else
{
    respawnPosition = new Vector3(0, 0, 0);
}
```

A similar error appeared because the save file wasn't working – it was of the loaded checkpoint being a null value. This broke the game, as a save file would perhaps also be necessary without a checkpoint (e.g. at the start of level 2 or 3). This would catch the null and set the respawn position back to the initial value, therefore deactivating any checkpoints as well.

```
public void Save(ref PlayerSaveData data)
GameManager.instance.Player.GetComponent<Player>().Save(ref _saveData.PlayerData); // call save in Player
```

To fix the save issue, I went back over the tutorial and realised I hadn't been using the ref keyword for the saving. I added it back in and, after running, determined that it was working.

```
string fileName = Application.persistentDataPath + "slot1" + ".save";
if (!checkSaveEmpty(fileName))
{
    overwriteCheck(3);
}
else
{
    GameManager.instance.SaveSlot = 3;
    SceneManager.LoadScene("Gameplay");
}
```

Now that I had some save data to work with, I went back to the New Game Menu. I realised that although it was showing that the save was not empty, the overwrite popup was not appearing and thus the saves were being immediately overwritten. This is because the overwrite popup was appearing, immediately disappearing, and then the save was overwritten nonetheless. To make the overwrite popup matter, I included the 'load' section in an else statement – this way, it wasn't trying to call the popup and then immediately continuing into the load.

```
3 references
public void FadePopup(bool fadeBool)
{
    selectBuffer = 2;
if (selectBuffer > 0)
selectBuffer -= 1;
if (selectedButton == 0 && selectBuffer <= 0) // overwrite
else if (selectedButton == 1 && selectBuffer <= 0)
```

The reason it was immediately disappearing was because it was calling the popup, and then the button was immediately clicked. By adding a selectBuffer attribute, the buttons cannot be clicked immediately. This attribute is reset every time it fades in or out, to ensure it always takes effect.

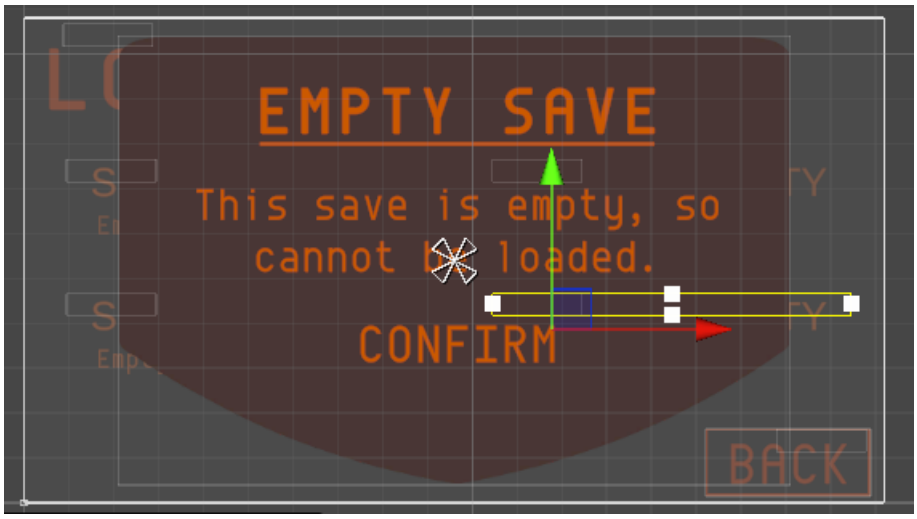
```
data.time = System.DateTime.UtcNow.ToString("HH:mm dd MMMM, yyyy"); // save the current value of the time
```

Using the Unity Discussions Board for help, I got the time and saved it to the file.

```
for (int n = 1; n <= 4; n++)
{
    string fileName = Application.persistentDataPath + "slot" + n + ".save";
    if (checkSaveEmpty(fileName))
    {
        GameObject.Find("Save Slot " + n).GetComponent<TextMeshProUGUI>().text = "SLOT " + n + " - EMPTY";
        GameObject.Find("Save Details " + n).GetComponent<TextMeshProUGUI>().text = "Empty Save";
    }
    else
    {
        string time = SaveSystem.readFile(n);
        if (time == null)
            time = "N/A";
        GameObject.Find("Save Slot " + n).GetComponent<TextMeshProUGUI>().text = "SLOT " + n + " - "; // add level
        GameObject.Find("Save Details " + n).GetComponent<TextMeshProUGUI>().text = "Last Saved: " + time;
    }

}
```

```
1 reference
public static string readFile(int fileNumber)
{
    string saveFile = Application.persistentDataPath + "slot" + fileNumber + ".save";
    string saveContent = File.ReadAllText(saveFile);
    _saveData = JsonUtility.FromJson<SaveData>(saveContent);
    return _saveData.PlayerData.time;
}
```

I significantly shortened the ShowData() method in New Game Menu, and added a new method in the SaveSystem to read the required parts of the file. This would print the time saved of each file and, later, the level saved on.



I added all the objects from the same menu to the load game menu. I replaced the text on the popup with a reminder that the user cannot load from an empty save.

```
if (selectedButton == 0) // slot 1
{
    string fileName = Application.persistentDataPath + "slot1" + ".save";
    if (checkSaveEmpty(fileName))
    {
        popup();
    }
    else
    {
        GameManager.instance.SaveSlot = 1;
        SaveSystem.Load();
    }
}
```

Although most of the Load Game menu will be copied from the New Game menu, some things need to be changed. The first was the selection of a file – if it was not empty, it would load the game. If not, it would call the popup. Thus, the correct scene must be called. This will be implemented after this menu is completed.

```
4 references
private void popup()
{
    GameObject popup = GameObject.Find("Empty Save Popup");
    popup.GetComponent<EmptySavePopup>().FadePopup(true);
    emptyPopup = true;
}
```

This doesn't have to be as complex, as the popup just appears and disappears. This also means that it does not need this its own script, as it can be handled in the menu.

```
else
{
    selectBuffer -= 1;
    if ((Input.GetKeyDown(KeyCode.E) || Input.GetKeyDown(KeyCode.F)) && selectBuffer <= 0)
    {
        GameObject popup = GameObject.Find("Empty Save Popup");
        popup.GetComponent<EmptySavePopup>().FadePopup(false); // hide
        emptyPopup = false; // cannot be clicked
    }
}
```

This just uses code similar to that in SaveOverwrite to buffer the ability to click the button, and disappear the menu when done so. This only works so easily because there is only one button.

```
0 references
void Start()
{
    FadePopup(false);
}

3 references
public void FadePopup(bool fadeBool)
{
    selectBuffer = 2;
    foreach (Transform child in transform)
    {
        try // if has  UI
        {
            StartCoroutine(child.gameObject.GetComponent<FadeUI>().Fade(fadeBool));
        }
        catch // if doesn't have UI (must have text)
        {
            StartCoroutine(child.gameObject.GetComponent<FadeText>().Fade(fadeBool));
        }
    }
```

The above means that the EmptySavePopup script only needs to be able to fade the popup.

```
data.currentScene = SceneManager.GetActiveScene().name; // save the name of the current scene
```

```
1 reference
private static void HandleLoadData()
{
    Debug.Log(SaveFileName());
    if (SceneManager.GetActiveScene().name != _saveData.PlayerData.currentScene && _saveData.PlayerData.currentScene != null)
    {
        SceneManager.LoadScene(_saveData.PlayerData.currentScene);
    }
```

I allowed the player to save the scene, and the SaveSystem to load it (as long as there is one).

```
data.time = System.DateTime.UtcNow.ToString("HH:mm dd/MM/yy");
```

```
GameObject.Find("Save Details " + n).GetComponent<TextMeshProUGUI>().text = "Saved: " + time;
```

I changed the format of the time and the save details, so that it would be presented in a more aesthetically pleasing manner.

```
if (player == null && GameObject.FindWithTag("Player") != null)
{
    player = GameObject.FindWithTag("Player");
}
```

This exists within the GameManager's Update, ensuring that the player value always has a gameObject assigned to it as long as the player exists.

```
1 reference
public static string readFileScene(int fileNumber)
{
    string saveFile = Application.persistentDataPath + "slot" + fileNumber + ".save";
    string saveContent = File.ReadAllText(saveFile);
    _saveData = JsonUtility.FromJson<SaveData>(saveContent);
    return _saveData.PlayerData.currentScene;
}
```

I created a new method in SaveSystem to return the scene in the save (and renamed the one that returns time to be more consistent with the name scheme shown above.

```
0 references
void Awake()
{
    body = GetComponent<Rigidbody2D>(); // get the reference to the Rigidbody2D
    GameManager.instance.Player = gameObject;
}

0 references
void Start()
{
    if (SaveSystem.readFileScene(GameManager.instance.SaveSlot) == SceneManager.GetActiveScene().name) // if loading t
    {
        SaveSystem.Load();
    }
}
```

I then used that in the Start() of the player to determine whether the scene that was just created was loaded from the save file, and to load the game if so. I chose to put it in Start() so I could ensure that the GameManager could have the gameObject of the Player assigned to it properly before it tried to load again.

```
respawnPosition = data.checkpointPosition; // set the respawnPosition to the one in the file
Debug.Log(data.checkpointPosition + " E");
if (respawnPosition == null || respawnPosition == new Vector3(0, 0, 0)) // if null or no checkpoint
{
    respawnPosition = new Vector3(0, 0, 0);
    transform.position = respawnPosition;
}
else
{
    transform.position = respawnPosition;
    findCheckpoint = true;
}
```

I realised after running a few times that the reason the load wasn't working properly was because the ID of the checkpoint was changing between runs, so it could not use the gameObject of the active checkpoint. I changed it to the position of the checkpoint, which allowed the user to go the correct position with no errors, as that was consistent no matter what.

```
0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag(HAZARD_TAG) && isAlive)
    {
        StartCoroutine(playerDeath());
    }
    if (collision.name == "Checkpoint" && findCheckpoint == true)
    {
        activeCheckpoint = collision.gameObject; // set touching checkpoint as the active one
        activeCheckpoint.GetComponent<Checkpoint>().activateCheckpoint();
        findCheckpoint = false; // no longer need to find a checkpoint
    }
}
```

Because the checkpoint that was loaded to wasn't activated when the game was loaded, I wrote this piece of code to use the checkpoint that was being touched initially to get the correct GameObject and activate that checkpoint.

```
data.time = System.DateTime.Now.ToString("HH:mm dd/MM/yy"); // save the current value of the time
```

I also changed the time just to be system time, rather than UTC, as it makes more sense to the user.

```
GameManager.instance.SaveSlot = saveSlot;
string fileName = Application.persistentDataPath + "slot" + saveSlot + ".save";
Debug.Log(fileName);
if (File.Exists(fileName))
{
    File.Delete(fileName); // delete previous file
}
SceneManager.LoadScene("Gameplay");
```

```
if (!File.Exists(Application.persistentDataPath + "slot" + GameManager.instance.SaveSlot + ".save"))
{
    SaveSystem.Save();
}
```

Finally, I made it so that the overwrite would actively delete the file. I did this last because it would have been more irritating in previous stages to have to continuously make and overwrite files. The player saves a new file immediately (in Awake) in order to create a file with that path – without, it would create errors.

Testing

Test No.	Description	Type	Data	Expected Result	Actual Result
1	NewGameMenu visible	Valid	Select New Game on Main	Menu is visible and unobscured by the popup	Stage 6.mp4 00:02
2a	Empty slots presenting as required	Valid	New Game Menu	Empty saves read "SLOT n – EMPTY SAVE", with no further details	Stage 6.mp4 00:02
2b	Full slots presenting as required	Valid	New Game Menu	Full saves read "SLOT n – " and have the time in the details section	Stage 6.mp4 00:02
3a	Can use arrow keys	Valid	Press right arrow	Box selector moves right	Stage 6.mp4 00:03
3b	Can use arrow keys	Valid	Press left arrow	Box selector moves left	Stage 6.mp4 00:03
3c	Can use WASD	Valid	Press 'D'	Box selector moves right	Stage 6.mp4 00:04
3d	Can use WASD	Valid	Press 'A'	Box selector moves left	Stage 6.mp4 00:04
4a	Overwrite popup appears when selecting a full save	Valid	Select Save 1	Popup appears	Stage 6.mp4 00:42
4b	Overwrite can be cancelled	Valid	Select Cancel	Popup disappears	Stage 6.mp4 00:43
4c	Can overwrite save	Valid	Select Overwrite	Goes to beginning of gameplay	Stage 6.mp4 00:45
4d	Overwrite popup does not appear on an empty save	Invalid	Select Save 2	Goes to beginning of gameplay	Stage 6.mp4 00:07
6	Checkpoint can save progress	Valid	Activate checkpoint, load save	Goes to that checkpoint in gameplay	Stage 6.mp4 01:29
7	Can overwrite a save	Valid	Reload game after overwriting save and reselect that save	Goes to beginning of gameplay	Stage 6.mp4 01:25
8	LoadGameMenu visible	Valid	Select Load Game on Main	Menu is visible and unobscured by the popup	Stage 6.mp4 01:21
9a	Empty slots presenting as required	Valid	New Game Menu	Empty saves read "SLOT n – EMPTY SAVE", with no further details	Stage 6.mp4 01:21
9b	Full slots presenting as required	Valid	New Game Menu	Full saves read "SLOT n – " and have the time in the details section	Stage 6.mp4 01:21
10a	Can use arrow keys	Valid	Press right arrow	Box selector moves right	Stage 6.mp4 01:23
10b	Can use arrow keys	Valid	Press left arrow	Box selector moves left	Stage 6.mp4 01:23

10c	Can use WASD	Valid	Press 'D'	Box selector moves right	Stage 6.mp4 01:24
10d	Can use WASD	Valid	Press 'A'	Box selector moves left	Stage 6.mp4 01:24
11	Save retains progress	Valid	Select Save 2	Goes to gameplay at Checkpoint 1	Stage 6.mp4 02:01

Evaluation

The aims of the stage were as follows;

- checkpoints save required game data to the requisite file
- game data includes current checkpoint and any upgrades
- new game select can create a new save file
- new game select can overwrite an existing save file
- load game can read the requisite file to start a game from that position
- save files persist beyond the game being terminated

I believe these were well met – a number of unforeseen problems emerged even with the use of a tutorial, and I think they were dealt with in a manner such that they will not pose any issues going forward. I am happy with the save system, and don't believe it will change in any further stage beyond adding in a few parts (like saving the level more accurately and registering that in the requisite screens).

Scene names may be subject to change, which is something to look out for in future stages.

References

GamePressure. (2025, 06 24). *How to easily get a good build in Dead Cells?* Retrieved from Game Pressure: https://www.gamepressure.com/dead-cells/how-to-easily-get-a-good-build/zfb35e

Motion Twin. (2025, 06 24). *Undying Shores, Dead Cells Wiki*. Retrieved from Dead Cells Wiki: https://deadcells.wiki.gg/wiki/Undying_Shores#/media/File:Undying_shores_Boat.png

Nintendo. (2025, 06 24). *Super Mario Bros. Wonder*. Retrieved from Super Mario Bros. Wonder, Nintendo Wiki: https://nintendo.fandom.com/wiki/Super_Mario_Bros._Wonder

Team Cherry. (2025, 06 24). *Knight, Hollow Knight Wiki*. Retrieved from Hollow Knight Wiki: https://hollowknight.fandom.com/wiki/Knight?file=Screenshot_HK_Hollow_Knight_Beta_25.png#Health_and_SOUL

Team Cherry. (2025, 06 24). *The Knight, Hollow Knight Wiki*. Retrieved from Hollow Knight Wiki: https://hollowknight.fandom.com/wiki/Knight?file=Screenshot_HK_Knight_10.png

Team Cherry. (2025, 06 24). *Troupe Master Grimm, Hollow Knight Wiki*. Retrieved from Hollow Knight Wiki: https://hollowknight.fandom.com/wiki/Grimm?file=Screenshot_HK_Grimm_07.png

Unity. (2025, 06 26). *Unity - Manual: System requirements for Unity 6.1*. Retrieved from Unity 6.1 User Manual: https://docs.unity3d.com/6000.1/Documentation/Manual/system-requirements.html#player