Lewis Wray

Contents

[Background and Project Outline 3](#_Toc41567077)

[Client/ End Users 3](#_Toc41567079)

[Alternate Systems 10](#_Toc41567080)

[Problem Modelling 14](#_Toc41567084)

[Data Structures 23](#_Toc41567091)

[Research 25](#_Toc41567088)

[Objectives 28](#_Toc41567089)

[Limitations 30](#_Toc41567090)

[Feasibility of potential solutions 31](#_Toc41567092)

# Background and Project Outline

# My project is a 2D platform-based shooter. I took gameplay inspiration from various games such as Hyper Jam and Stick Fight. I based the level design from other games like Super Crate Box and Move or Die. I like the pixel-graphics art-style used notably in the games Terraria, The Binding of Isaac, Celeste, Dead Cells, Super Meat Boy and Enter the Gungeon. I intend to merge elements from all these games in my project.

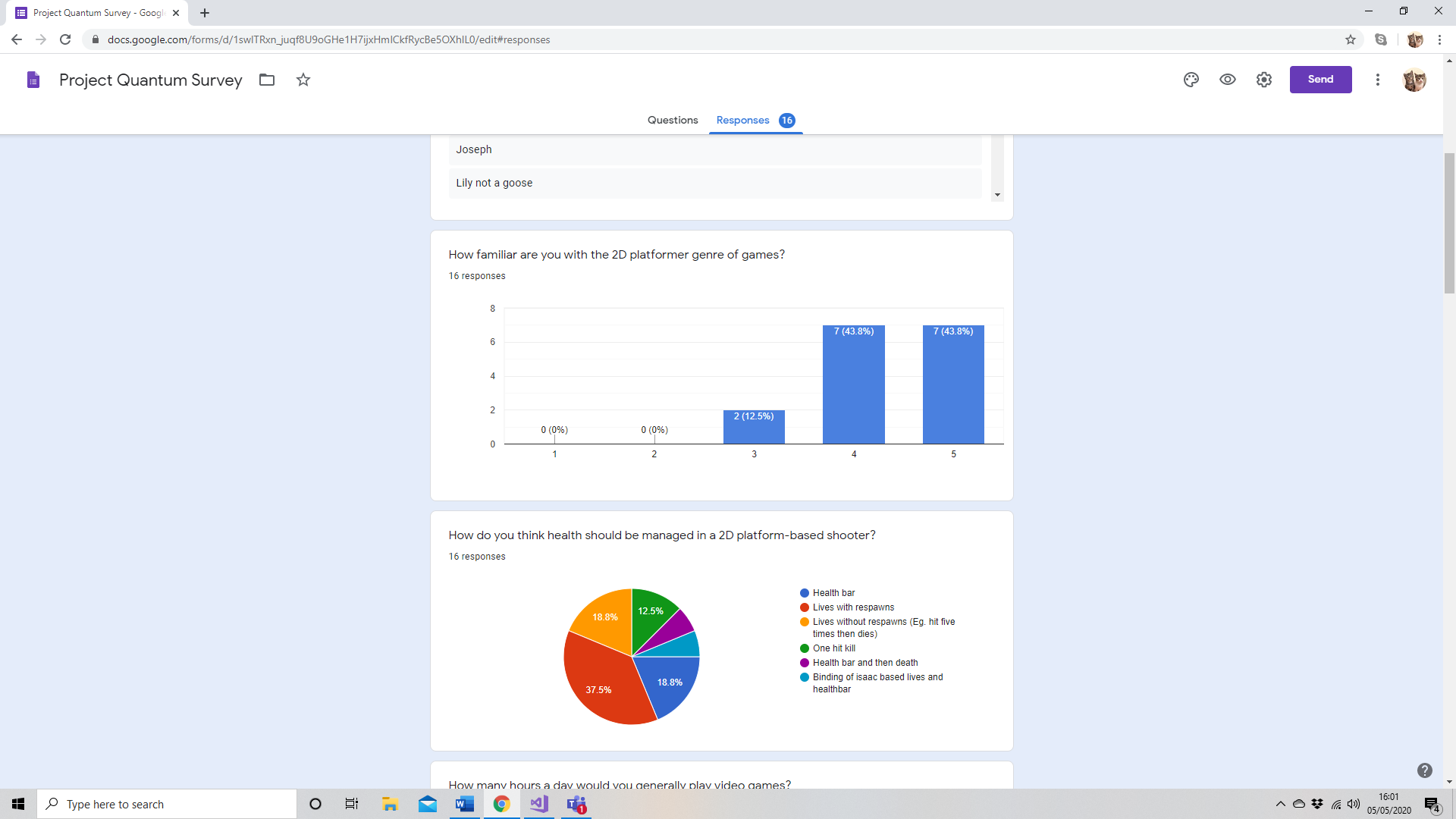
The game takes place in a weapons factory and will allow the player to control a character in a hazmat suit with a handgun. People break into the factory with the intent of stealing weapons. The lockdown procedure puts all the weapons into ‘quantum crates’, which spawn intermittently into the arena during the game. The player can collect these crates to get new weapons that do different things (e.g. flamethrower, grenade launcher). Enemies will spawn into the arena as well. They will have a melee attack that only hurts the player at very close range. The player’s aim will be to either last as long as possible against increasingly difficult waves of enemies or complete a set of defined enemies. These could be two different game modes. I could also add a two-player mode to the game where they fight each other.

Progressing further, I could add unlockables into the game. For example, there could be coins that spawn in the arenas for the player to pick up and use in the in-game shop. Here, the player could buy different characters, weapons, or cosmetic items. This would add a sense of progression to the game, as well as replayability value and it would provide more content for any players that would want to unlock everything they possibly can in a game.

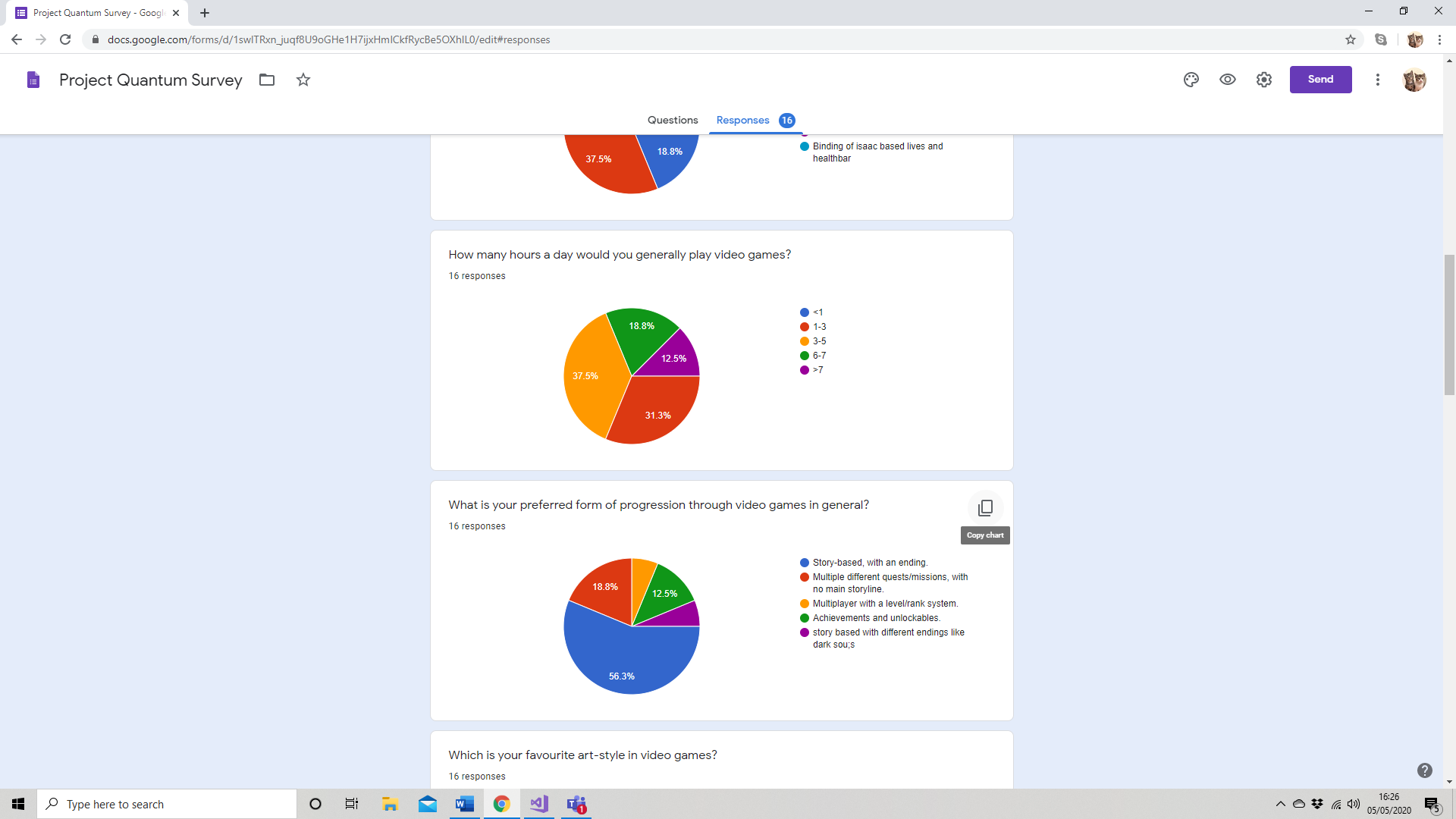
# Client/ End Users

The users of this game can be anyone interested in video games. It is not going a particularly difficult game to play at entry level, making it entirely accessible for a new player to get the hang of. As the game is supposed to be available for everyone, I must make sure the player is guided through well and is shown clearly how to play the game as well as how to progress and improve. I carried out a survey with potential users of the game. This included people who are very interested in games, as well as people who aren’t as involved to make sure my game can be approachable by people who aren’t very experienced in video games.

Survey: <https://forms.gle/S6yVvizjkQFeR2A87>

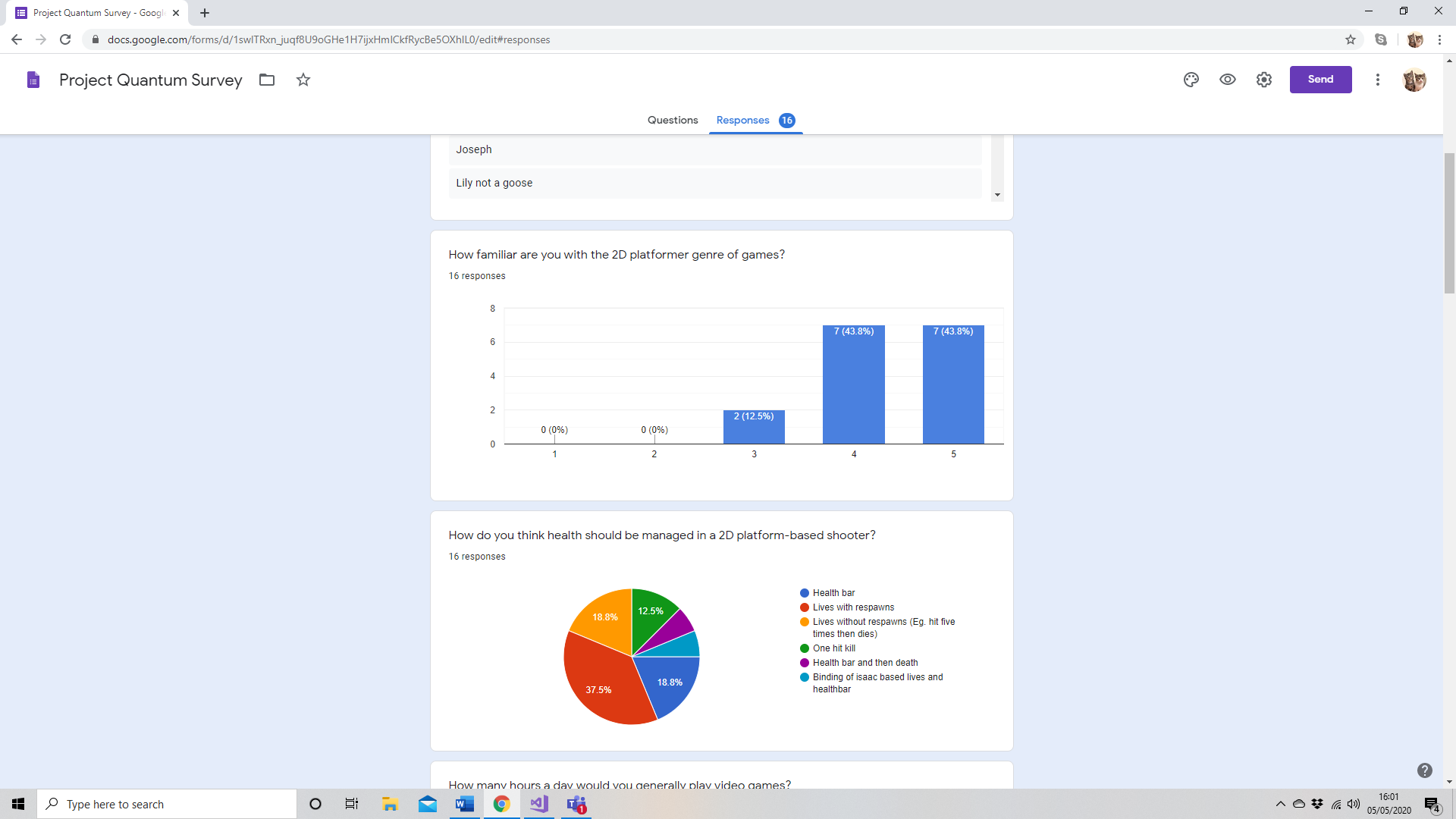


The first question determined roughly how many people knew about the genre of my game. The answers showed that most of the people taking the survey knew a lot about 2D platformers. This was reassuring not only because I knew I would be getting well informed answers and suggestions, but it also shows that most people – even if they don’t play video games often – are somewhat familiar with the genre. This means my 2D platformer game is open to a wide audience.



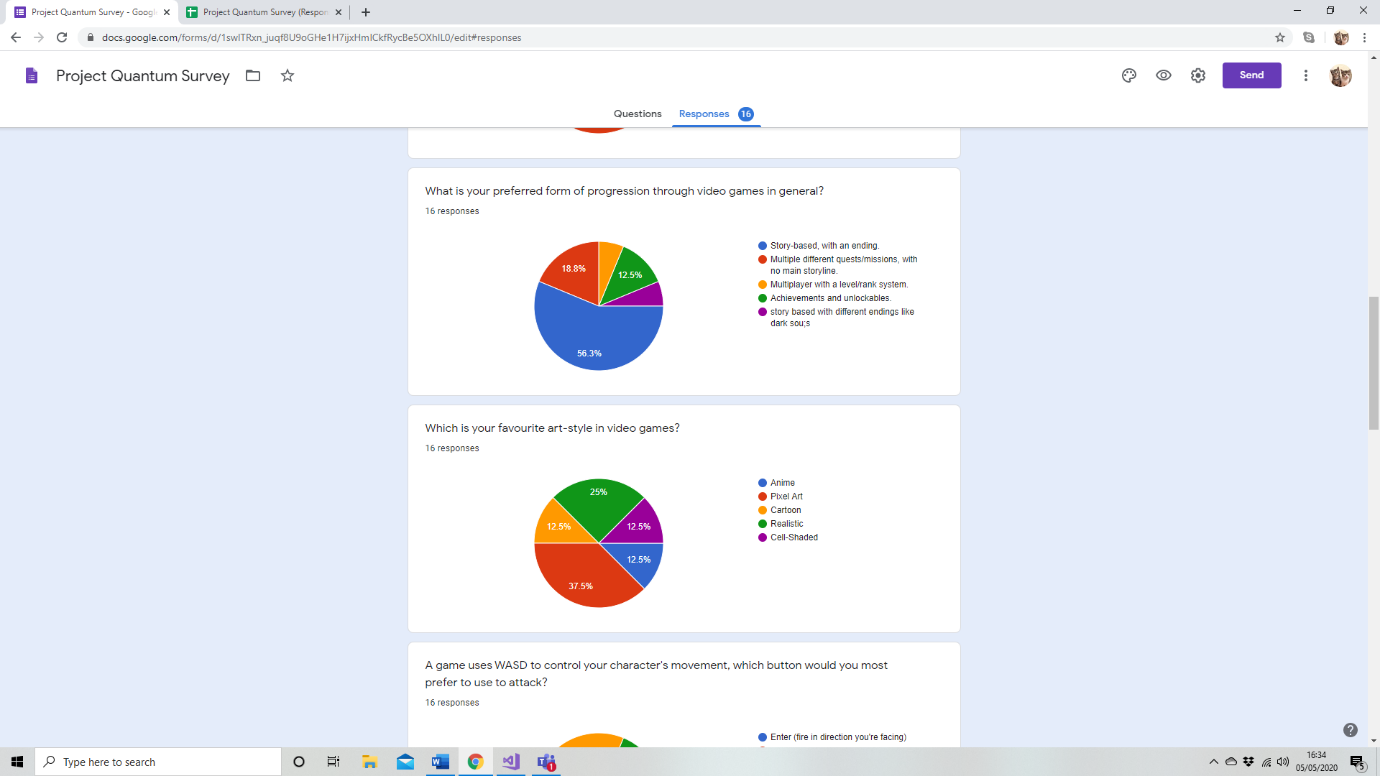
This question had a similar purpose to the first one. I wanted to gain some perspective on how familiar the people answering the questions were with video games. I decided that the time spent playing games per day is a good measurement. Luckily, I found quite an even spread, with most people playing around 3-5 hours a day. This just informs me that people with many different experiences of gaming are giving suggestions, which would make the responses as varied as possible.

This question had a similar purpose to the first one. I wanted to gain some perspective on how familiar the people answering the questions were with video games. I decided that the time spent playing games per day is a good measurement. Luckily, I found quite an even spread, with most people playing around 3-5 hours a day. This just informs me that people with many different experiences of gaming are giving suggestions, which would make the responses as varied as possible.

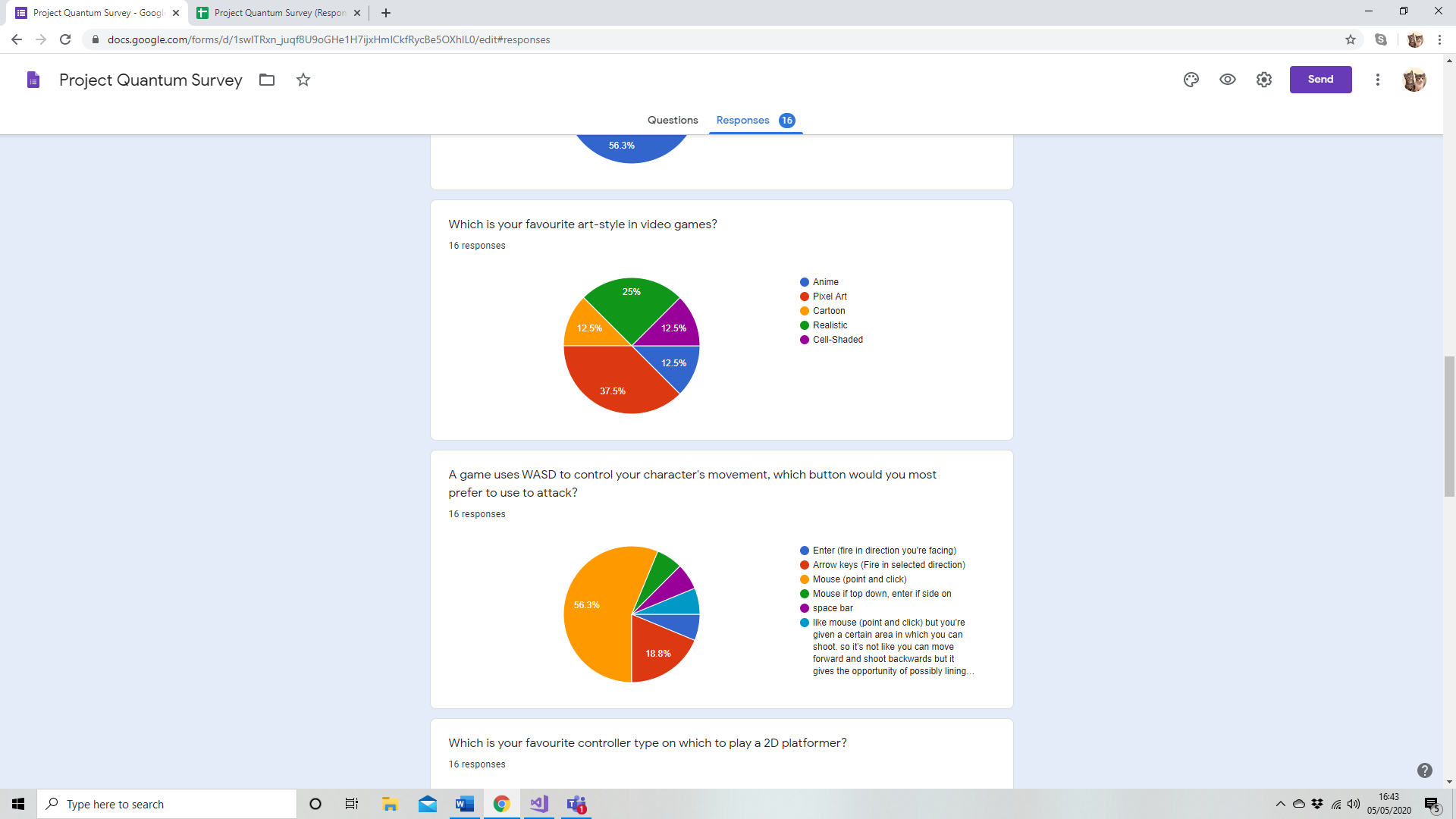


Over a third of people on this question suggested that the game should be managed by lives, with the player respawning and losing a life each time. This could work with my game as I could make it so the player has a certain number of lives in each level. This allows for implementation of ‘extra lives’ in the game to make it more interesting and give the player something else to try and pick up in each level. However, it removes the possibility for health boosts or any other health-based effects such as damage over time. Considering this, I could implement this with elements of another idea. For example, I could use a health bar with respawns.

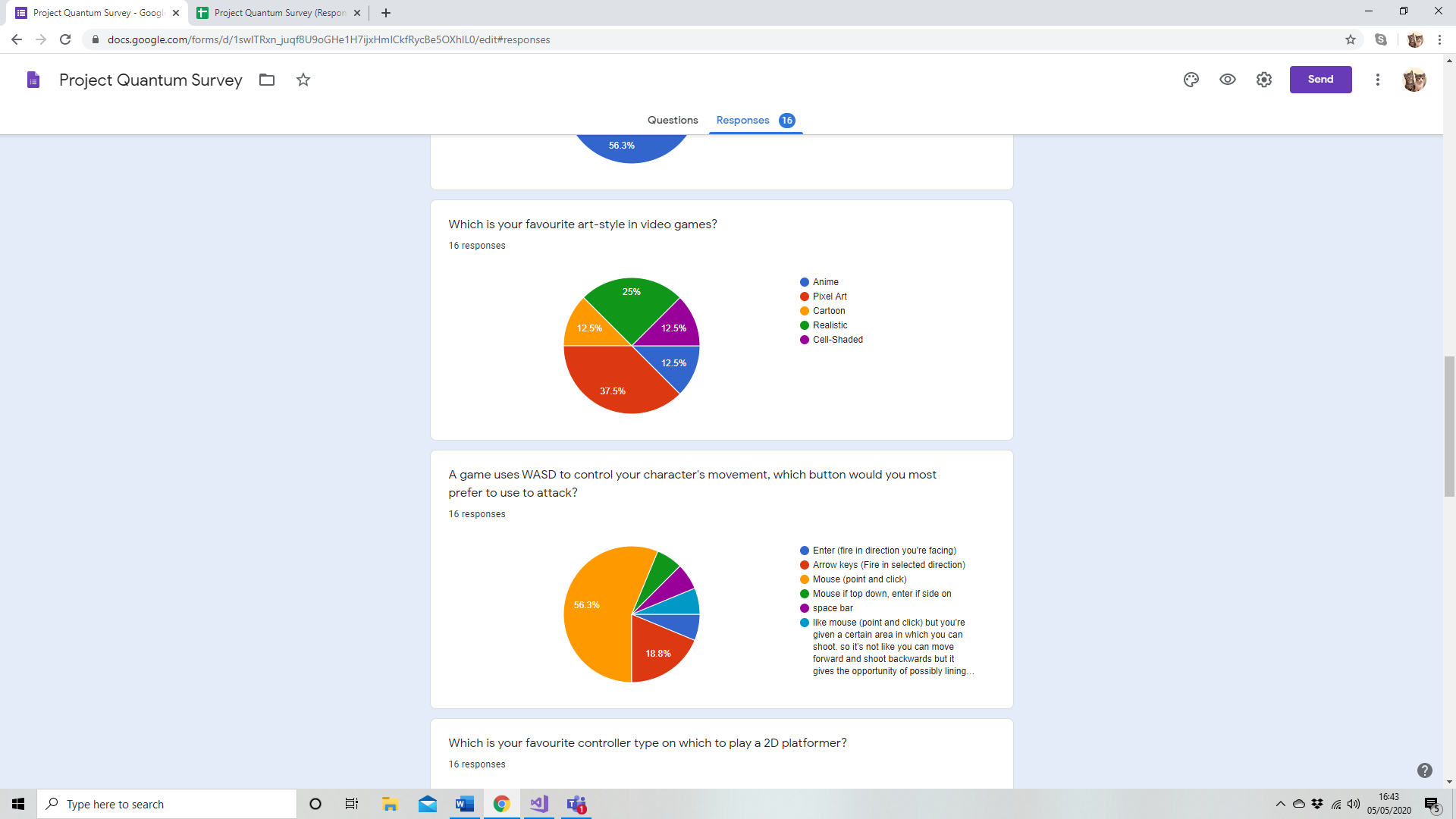
Over a third of people on this question suggested that the game should be managed by lives, with the player respawning and losing a life each time. This could work with my game as I could make it so the player has a certain number of lives in each level. This allows for implementation of ‘extra lives’ in the game to make it more interesting and give the player something else to try and pick up in each level. However, it removes the possibility for health boosts or any other health-based effects such as damage over time. Considering this, I could implement this with elements of another idea. For example, I could use a health bar with respawns.



When starting the project, I wasn’t sure what kind of direction my game should take. Luckily, this question gave me some guidance on people’s favourite kinds of progression. The overwhelmingly popular opinion is that games are best when they have a storyline and an ending (or multiple endings). I think this style could quite easily be in my game. I would have to design different levels and be sure to make them more difficult as the player progresses. There could be different endings; the player could choose which boss to fight at the end of the game, each one giving them a different reward.

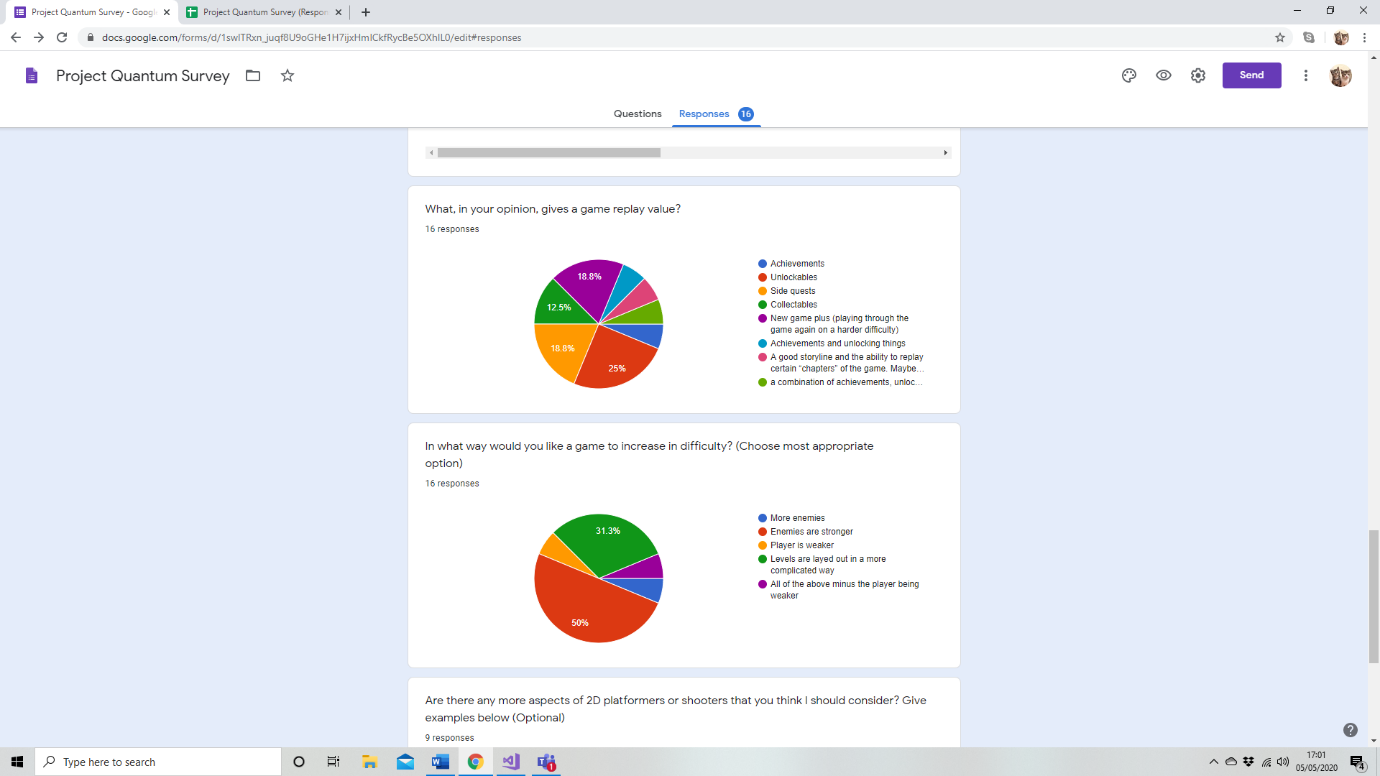


When asking this question, I was already leaning towards using pixel art in my game as it is easiest to animate and has a good 2D aesthetic. However, there was no harm in finding out how many people would be on board playing this style of game, as well as what style they would prefer. Luckily, the most common answer was pixel art, reassuring me that I had made the right decision. A quarter of people said their favourite is realistic. Although this style is very impressive, it might not work very well in a 2D platformer, not to mention the fact that it would require infinitely more time spent on art character and level design.

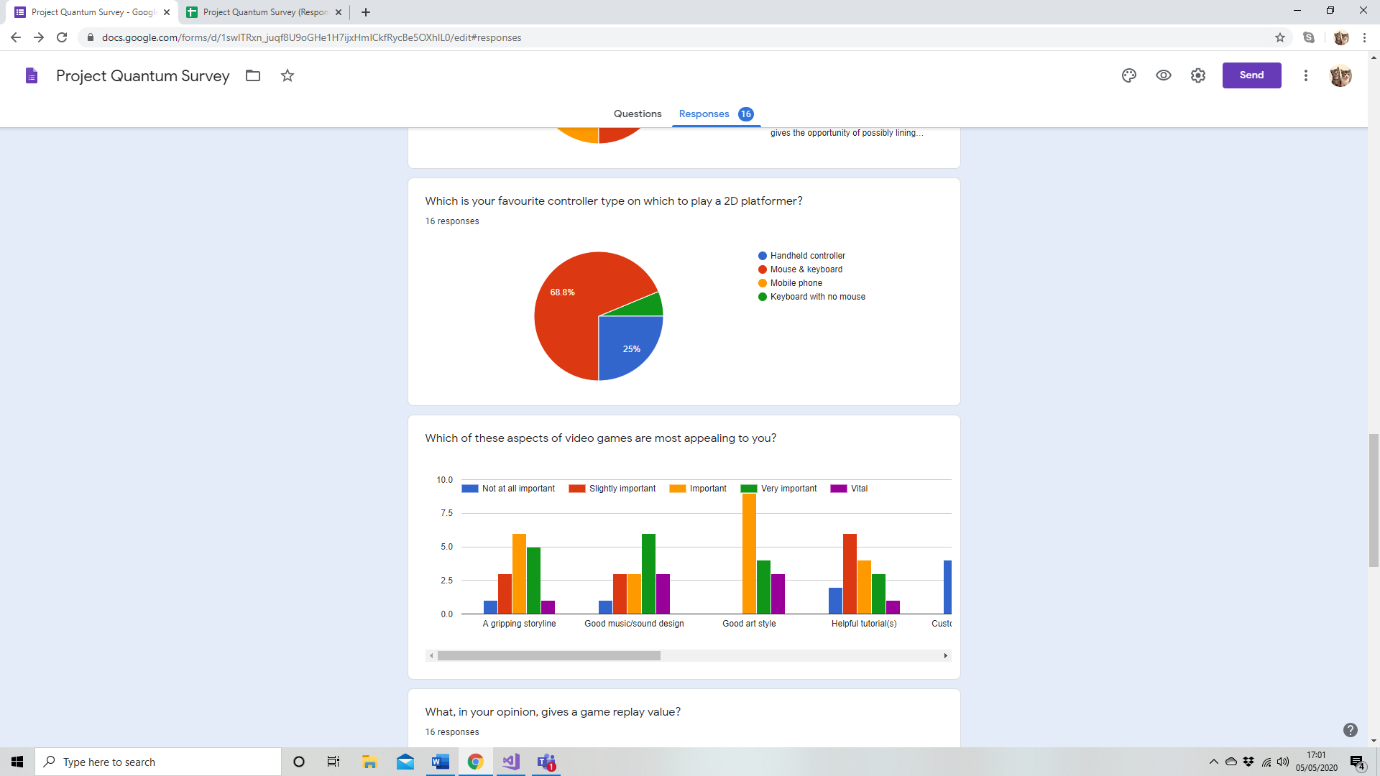


This was a very important question for me to ask, as the entire gameplay is based on shooting. Therefore, I had to determine the way in which most people would like to go about it. The majority of people said that using the mouse to click in the direction in which you want the character the shoot is the most effective option. I agree that this is a good mechanic as gives the player more freedom of where they can shoot. As well as this, the point-and-click system means the player has rely slightly more on their level of skill to be good at the game.

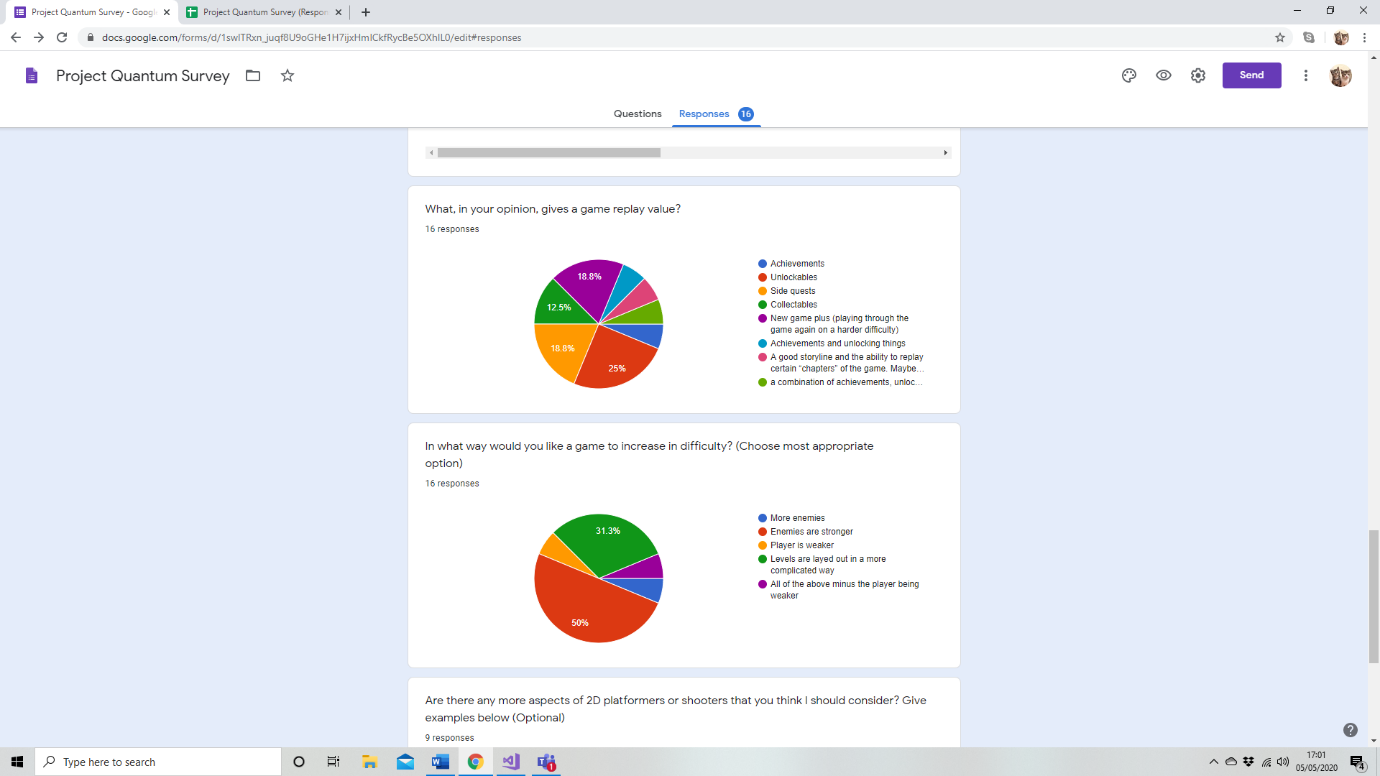
This was a very important question for me to ask, as the entire gameplay is based on shooting. Therefore, I had to determine the way in which most people would like to go about it. The majority of people said that using the mouse to click in the direction in which you want the character the shoot is the most effective option. I agree that this is a good mechanic as gives the player more freedom of where they can shoot. As well as this, the point-and-click system means the player has rely slightly more on their level of skill to be good at the game.

Unity provides support for controller use and mobile versions. I asked this question to determine whether I should add the option to play the game with a controller. This would require a slightly different style of play as the player would not be able to point and click to fire. They would have to use one of the analogue sticks on the controller to move and the other to aim, with a button or trigger to fire. Nobody said they would like to play the game on a mobile phone. I may make it available on mobile at some point, but it is certainly not a priority.

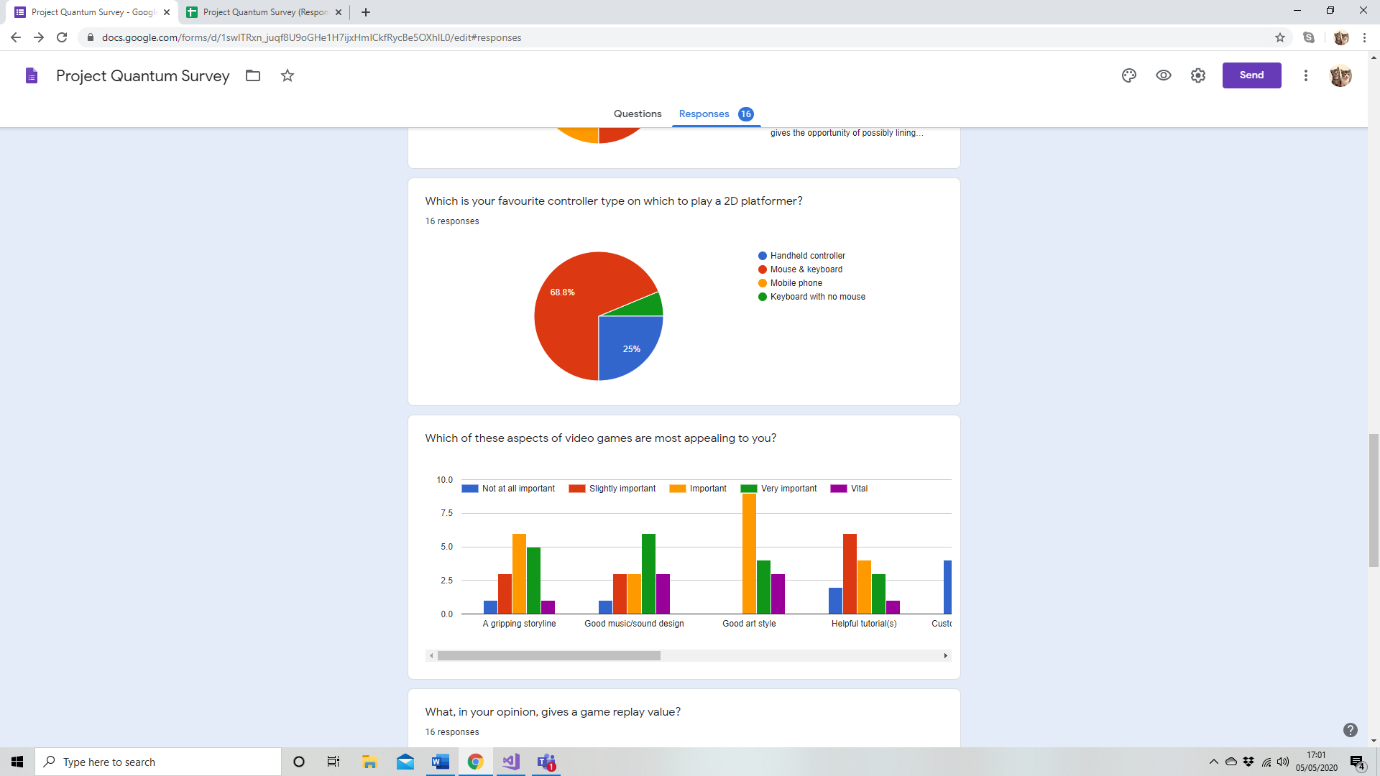
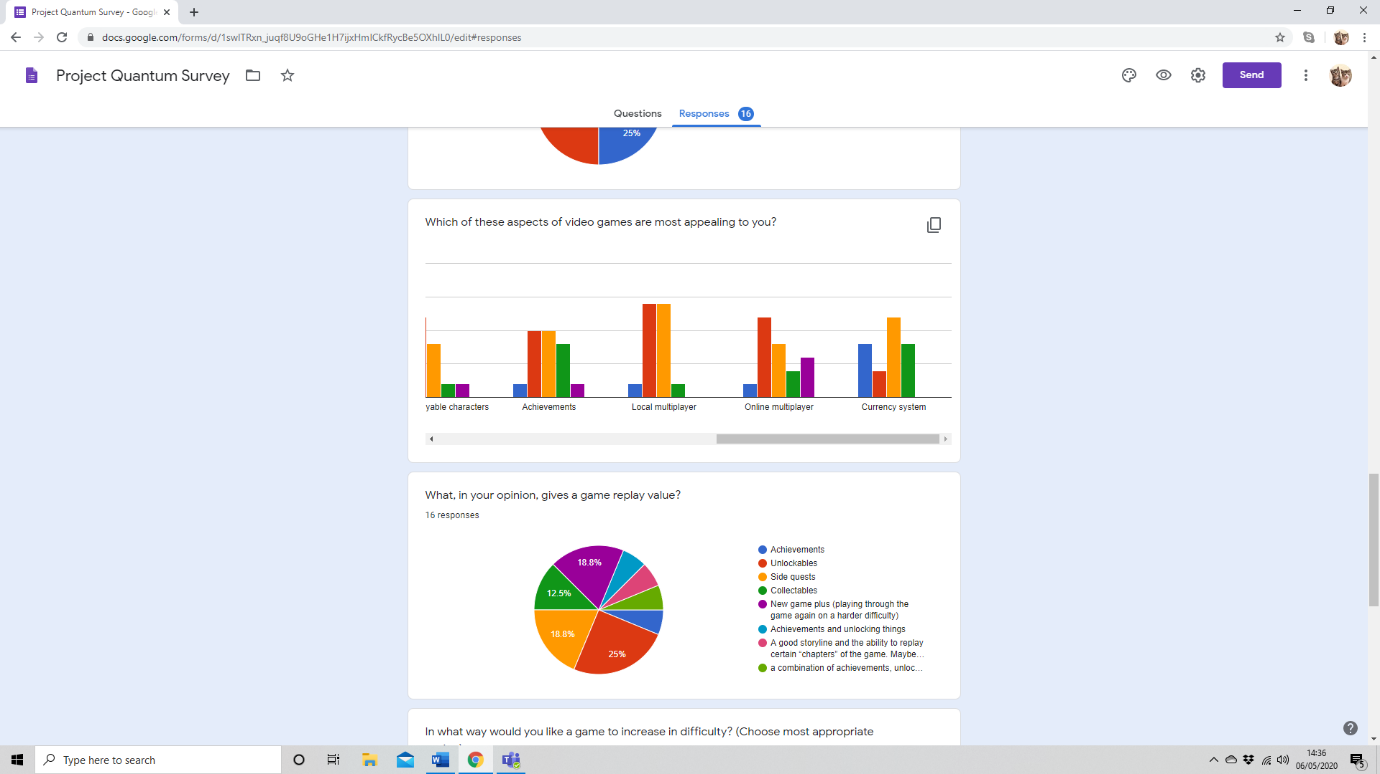
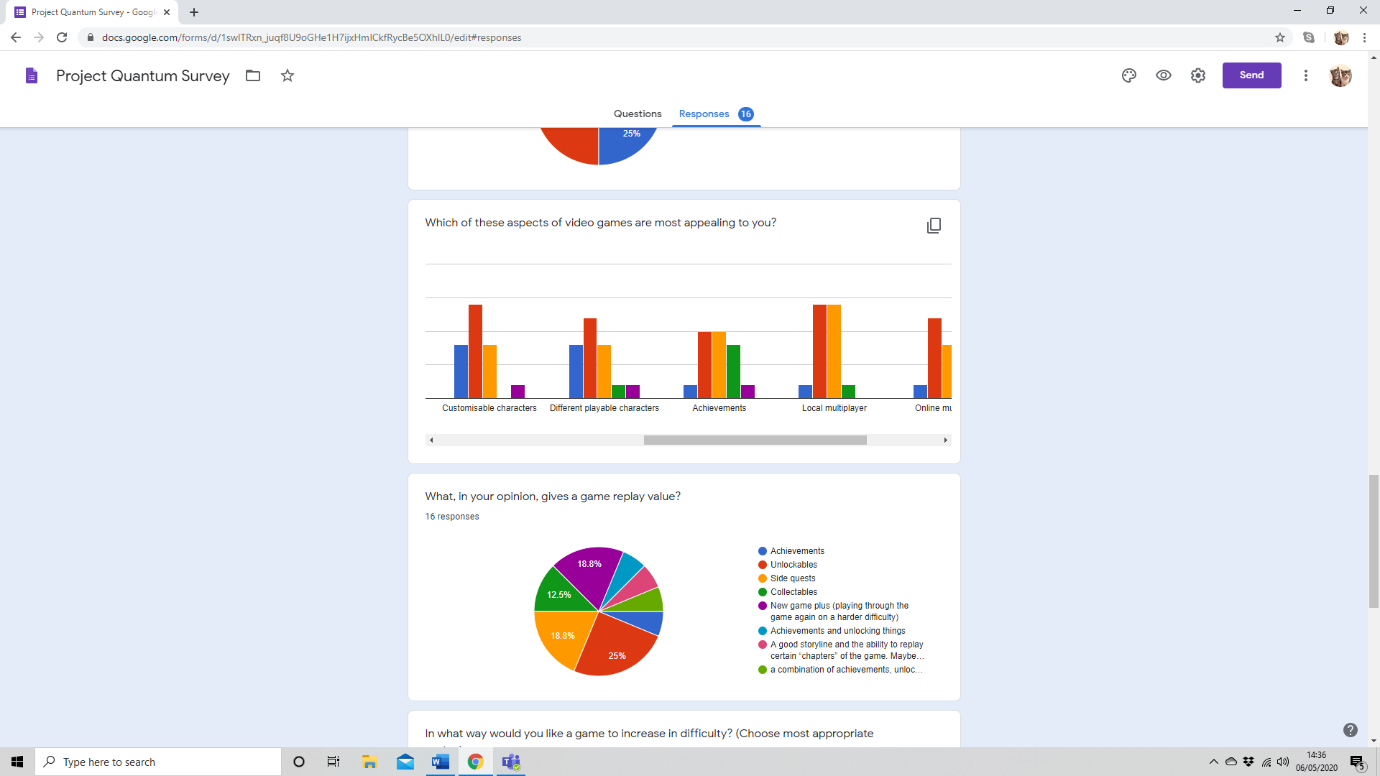
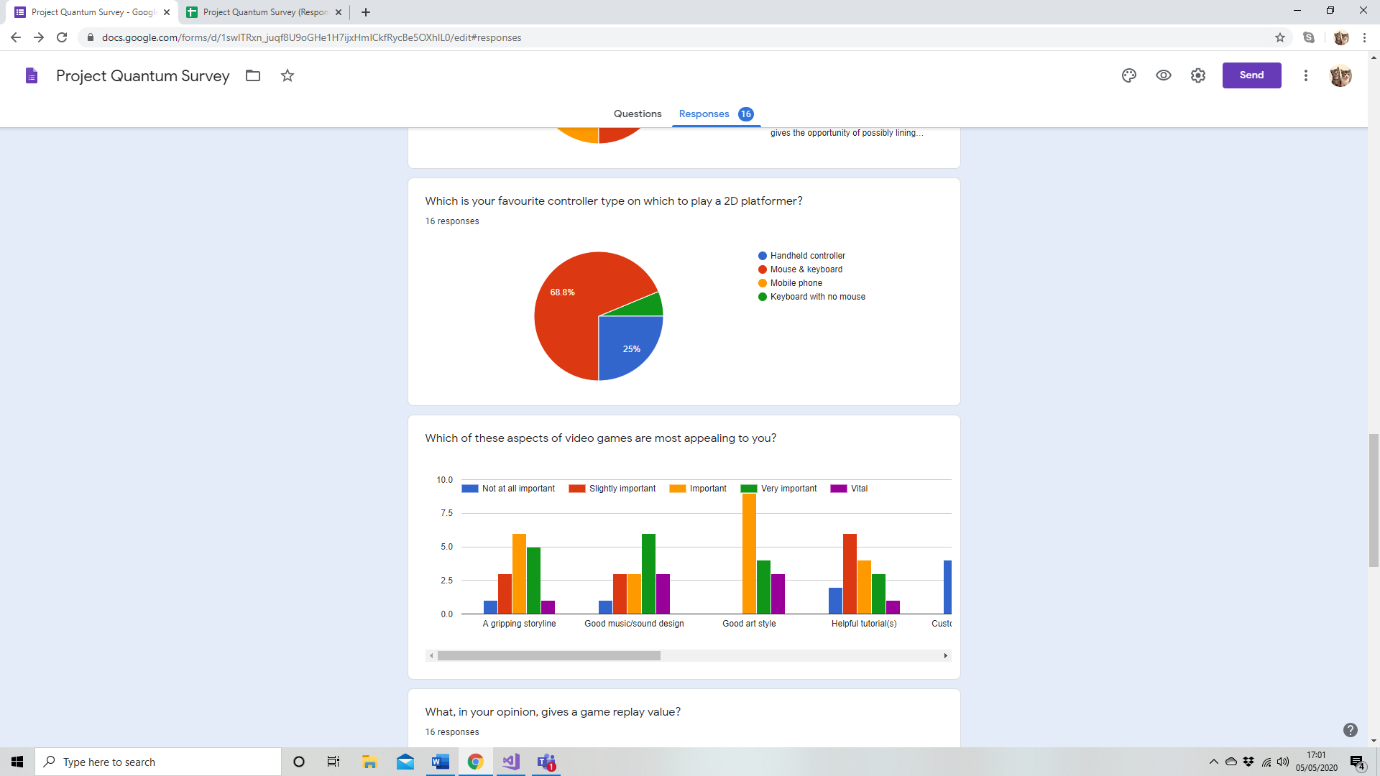
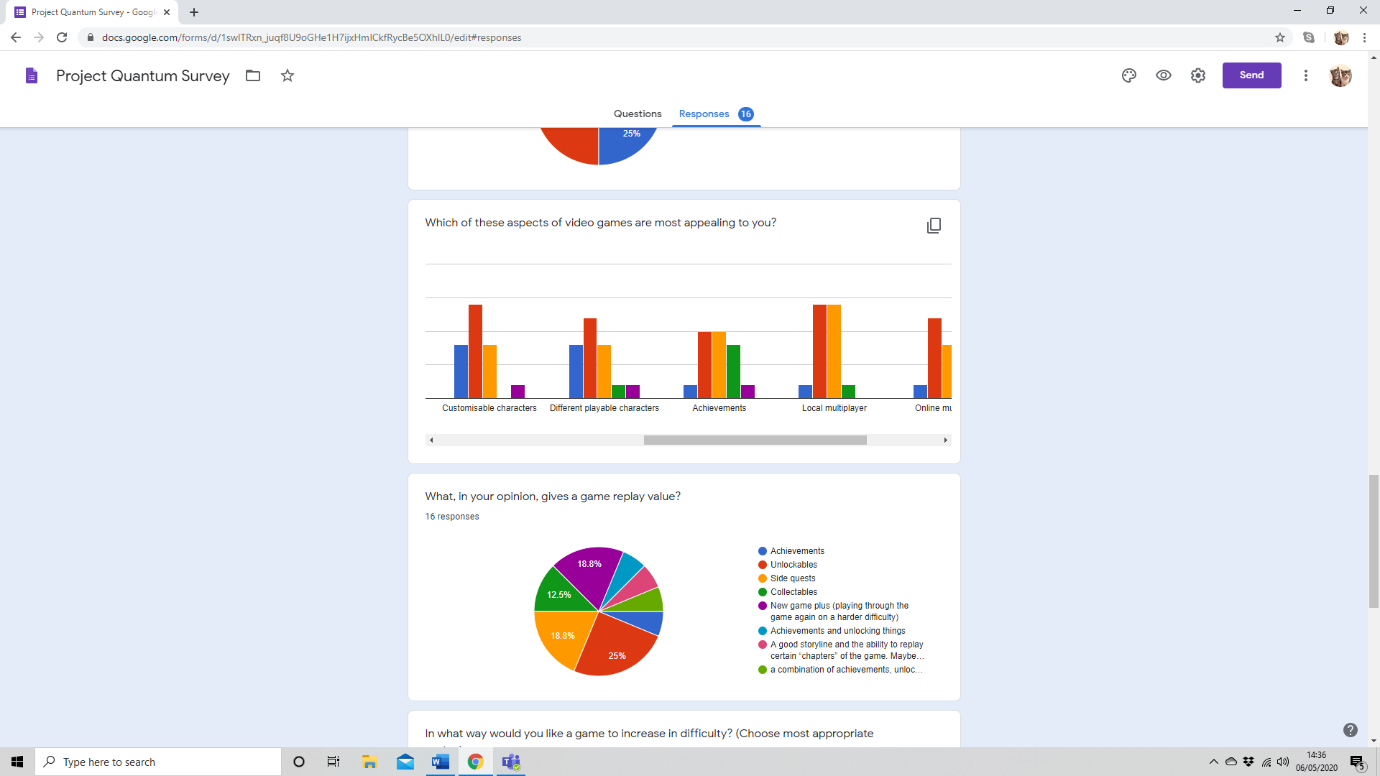
This question gave a wide variety responses. This is useful as I can work through adding them into my game in order of preference. I will start by adding unlockables into the game. This could be done through a points system in each level. If the player gets a certain amount of points in a level, they will unlock a new character to play as or a new outfit for their character. Alternatively, the player could accumulate some form of currency through playing the game, which they could use to buy things from an in-game shop.



Unity provides support for controller use and mobile versions. I asked this question to determine whether I should add the option to play the game with a controller. This would require a slightly different style of play as the player would not be able to point and click to fire. They would have to use one of the analogue sticks on the controller to move and the other to aim, with a button or trigger to fire. Nobody said they would like to play the game on a mobile phone. I may make it available on mobile at some point, but it is certainly not a priority.



Half the people answering this question suggested that enemies could get stronger as the game progresses. I could do something simple by making them faster or have more health. Or I could use completely different, more complex enemies in the different levels. Another popular suggestion was to make the structure of the levels themselves more complicated. This encourages the player to move around more skilfully in order to avoid the enemies and they will learn to use the environment to their advantage.



**A gripping storyline** – People appear to have very neutral feelings about this. One person thought it was vital, another thought it was useless, with most voting it around mid-table. Due to this, I might make a vague storyline, with sections of text between levels.

**Good music/ sound design** – People appear to have the opinion that good sound design is essential in games. I think I may investigate making a simple, retro, 8-Bit style soundtrack to play during the games, maybe getting more intense during a boss. I will also have different sound effects for different weapons, so they feel unique.

**Good art style** – This topic was already addressed in an earlier question. People seemed quite positive about the pixel graphics, which is good because they seemed to value a good art style quite highly.

**Helpful tutorials** – This part of the game apparently isn’t as important to people as others. A few people though it was quite important, but others think it isn’t necessary. Therefore, I might include an optional tutorial in the main menu.

**Customisable characters** – This was one of the aspects that people valued the lowest. Maybe people are more interested in playing the game, rather than making their character look nice. This is helpful to consider as I should spend less time designing outfits for the player.

**Different playable characters** – According to the survey, different playable characters are not that important in a game. Maybe I should only have one character to play as but make it so the player can change that character’s properties and abilities.

**Achievements** – This got a very neutral response. I think I might implement achievements into the game. They might just be for completing levels and reaching certain scores. These achievements could also reward players with currency or unlockable items to customise their character with. Whether that is abilities or cosmetics.

**Local multiplayer** – This element had a very negative response. It seems as if people aren’t as keen on local multiplayer as they are on a good single player or an online multiplayer. It may be for the best as a computer only has one mouse, making it difficult for two people to play at once

**Online multiplayer** – This response was a lot more positive than local multiplayer. Apparently, people prefer to play online. This would be very difficult to implement into my game, but it may be possible if the game was downloaded on two different devices.

**Currency system** – Nobody said a currency system is vital in a game. Maybe because it simply isn’t the most fun element of any game. I think it will be in my game nonetheless as it will control how much the player can unlock at each point and can be used as a form of progression. As well as this, it gives players something else to aim for in levels if there were coins spread around the arenas or if enemies dropped coins

# Alternate Systems

Game 1 – Super Crate Box

Super Crate Box is a game available on PC and Mac (previously mobile) in which the player must open crates to earn points. Each crate contains a random weapon that the player gets when they open the crate. They can then use these weapons to kill the enemies spawning from the top of the screen. The enemies very simply move forward and turn around when they collide with a wall. If an enemy reaches the bottom of the screen and falls into the fire, they turn red and respawn at the top. These enemies move faster so are harder to kill. This ensures the player should not only focus on getting crates and gaining points, but also on killing enemies so they don’t get overrun by fast-moving ones. As well as this, there are different types of enemies facing the player such as the larger ones that have more HP and the little floating ones that are slower but harder to hit. As the player unlocks crates while playing the game, they unlock more and more weapons that are available to them in crates. They can also unlock new levels by getting to a certain score in the previous level.

I take a lot of inspiration from this game as the concept is very similar to mine. I like the way the scoring system is based on opening crates as it deters the player from sitting in the corner with a good gun and expecting to progress. The enemies I am planning on using in my game will fly toward the player instead of just running forward. However, as the player progresses and the game gets harder, I could add different types of enemies that are more challenging to face. As my game uses story-based levels for the main game mode, I will most likely allow the player to unlock more weapons as they complete each level. Or, perhaps, they could unlock a new weapon if they complete a level with a certain score or in a certain amount of time. Completing a level normally will just give them access to the next level.

A picture containing screenshot, sign

Description automatically generatedA screenshot of a video game

Description automatically generatedA close up of a sign

Description automatically generatedA screenshot of a video game

Description automatically generated

A picture containing monitor

Description automatically generated

Game 2 – Move or Die

Another game I take inspiration from with my project is Move or Die. It is a local and online multiplayer game on PC, Mac and PS4. The aim of the game is to survive longer than the other players. The game works on the mechanic where the less you move, the faster your health bar decreases. This encourages the players to carry on moving around the arena while also trying to stay alive, and (similarly to Super Crate Box) discourages them from sitting in a corner and waiting to win. There are many different game modes that are chosen at random in each game. They involve avoiding falling blocks, racing to the finish, holding a crown to gain points, and colouring block in the arena to colour them in order to finish the game with the most coloured blocks. The Move or Die scoring system is also very effective; it gives scaled points based on how well you did in the game. First to 50 points wins. The game also lets you customise your character with different colours and costumes that can be unlocked through playing the game.

The elements of this game I could implement into my own are mainly for if I was to add a multiplayer section. The different game modes and scoring system seem very useful and I feel as if they would work well with my style of game, only with different game modes more suited to the specific gameplay I’ve created. The scoring system would also be perfect with the game I am creating. There are also smaller aspects of this game that would work well in mine. For example, I think the brightly coloured health bars look good with the character next to each one to prevent confusion. Additionally, the character customisation and unlockable cosmetics in this game would be good to take inspiration from. As would the use of different colours for different players and different teams.

A screen shot of a video game

Description automatically generated

A picture containing building, toy, front, light

Description automatically generated

A picture containing green, road, truck, light

Description automatically generated

# A picture containing monitor, television, screen, display Description automatically generated

Game 3 – Hyper Jam

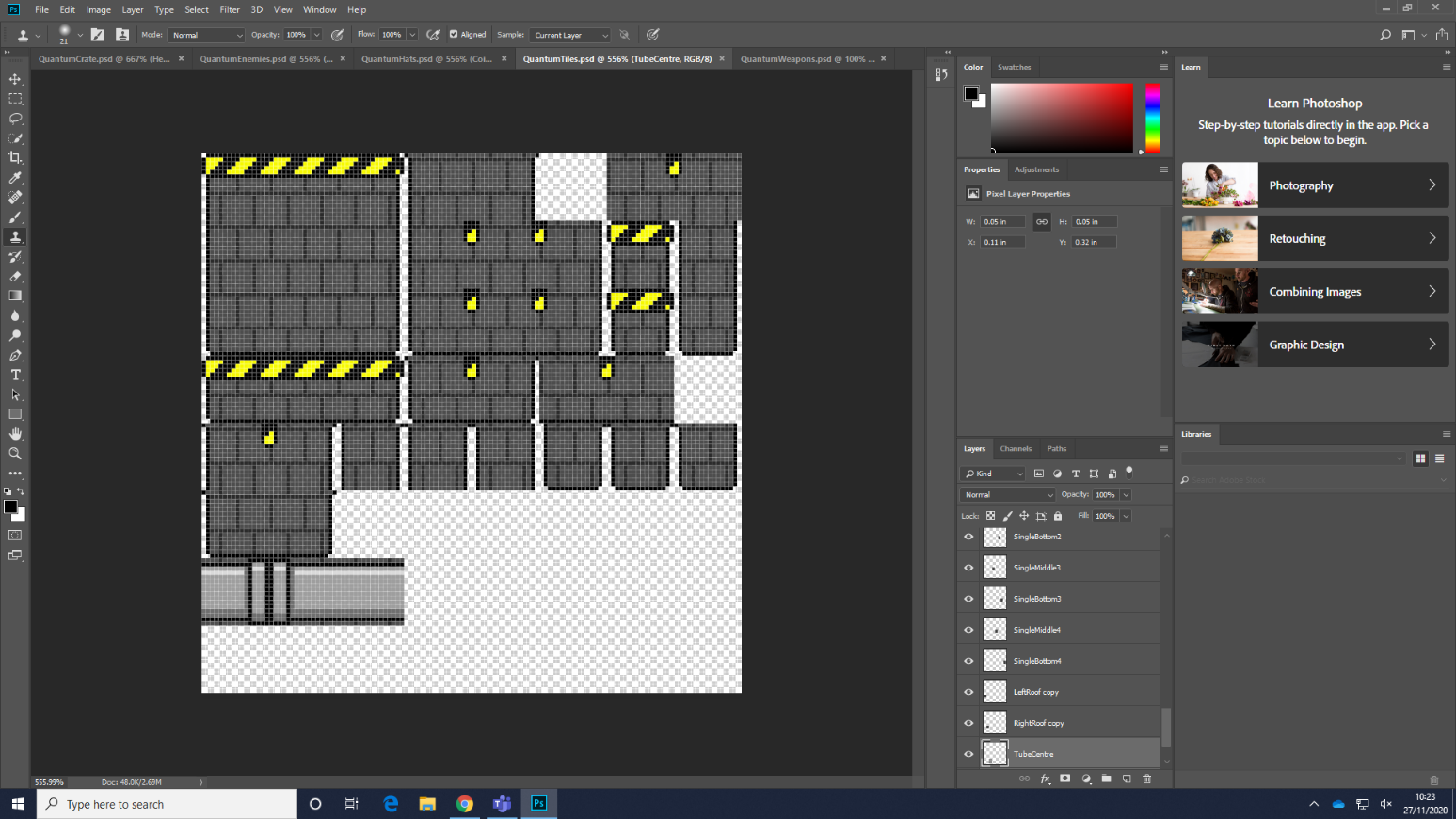
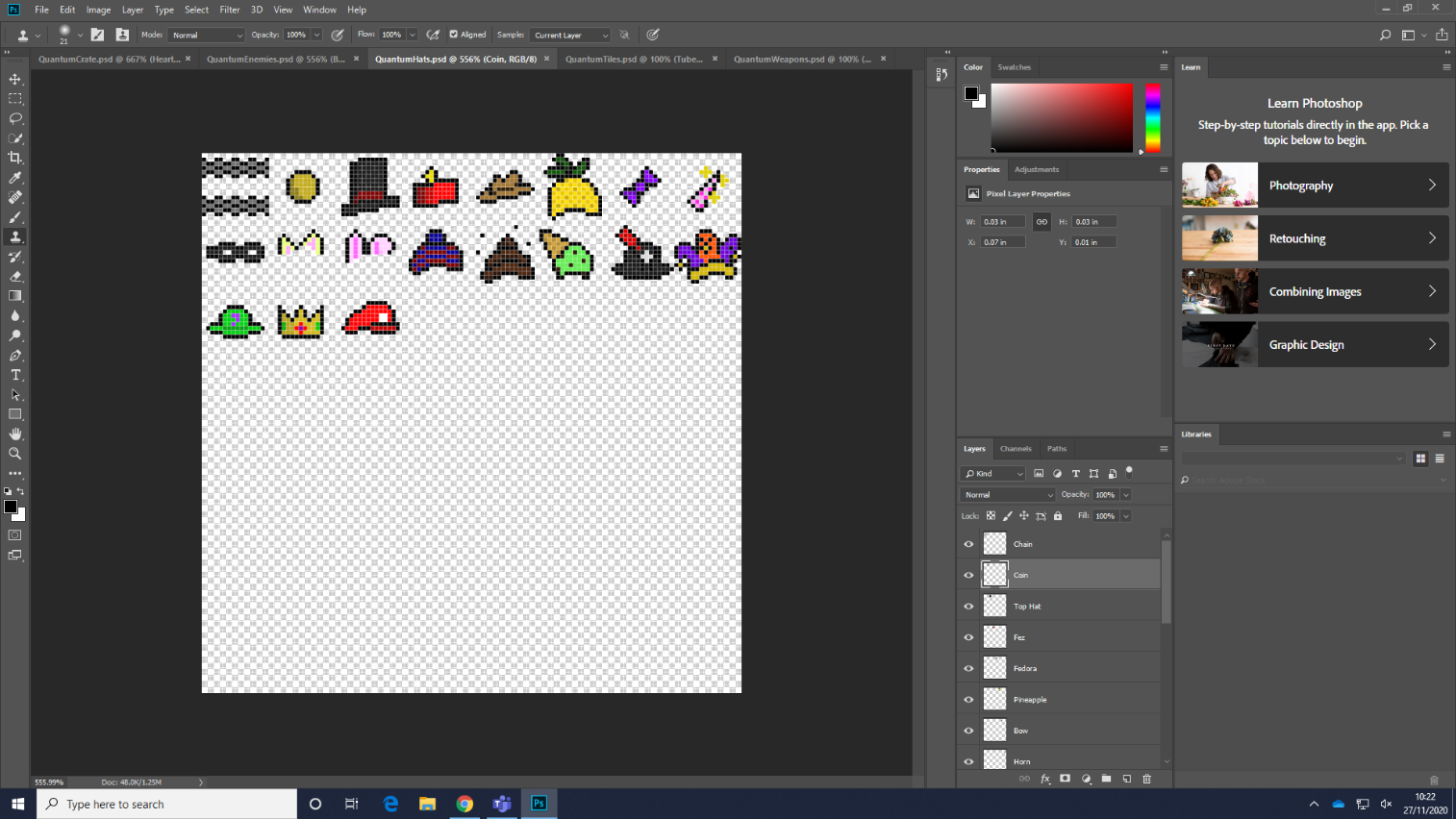
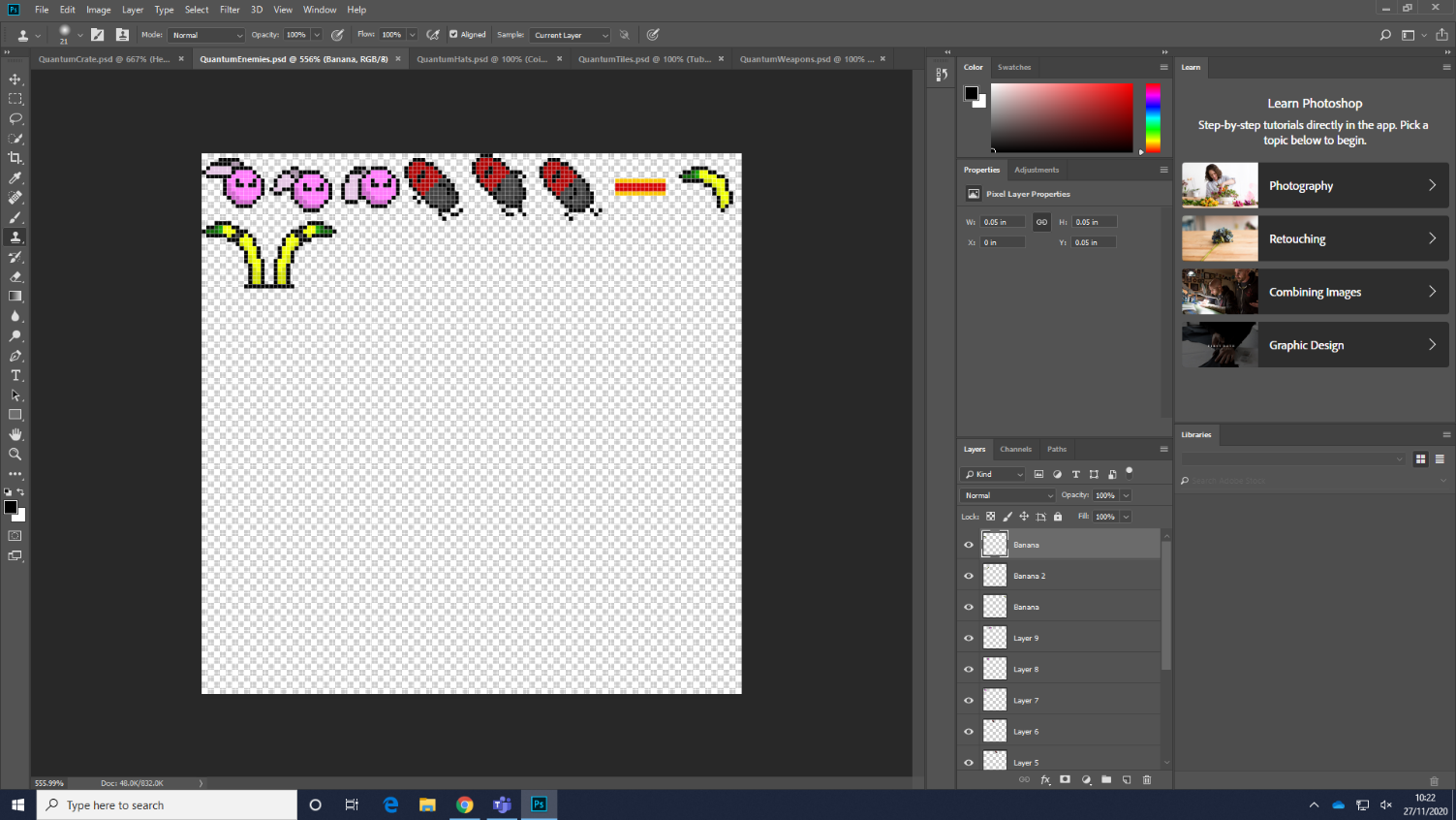
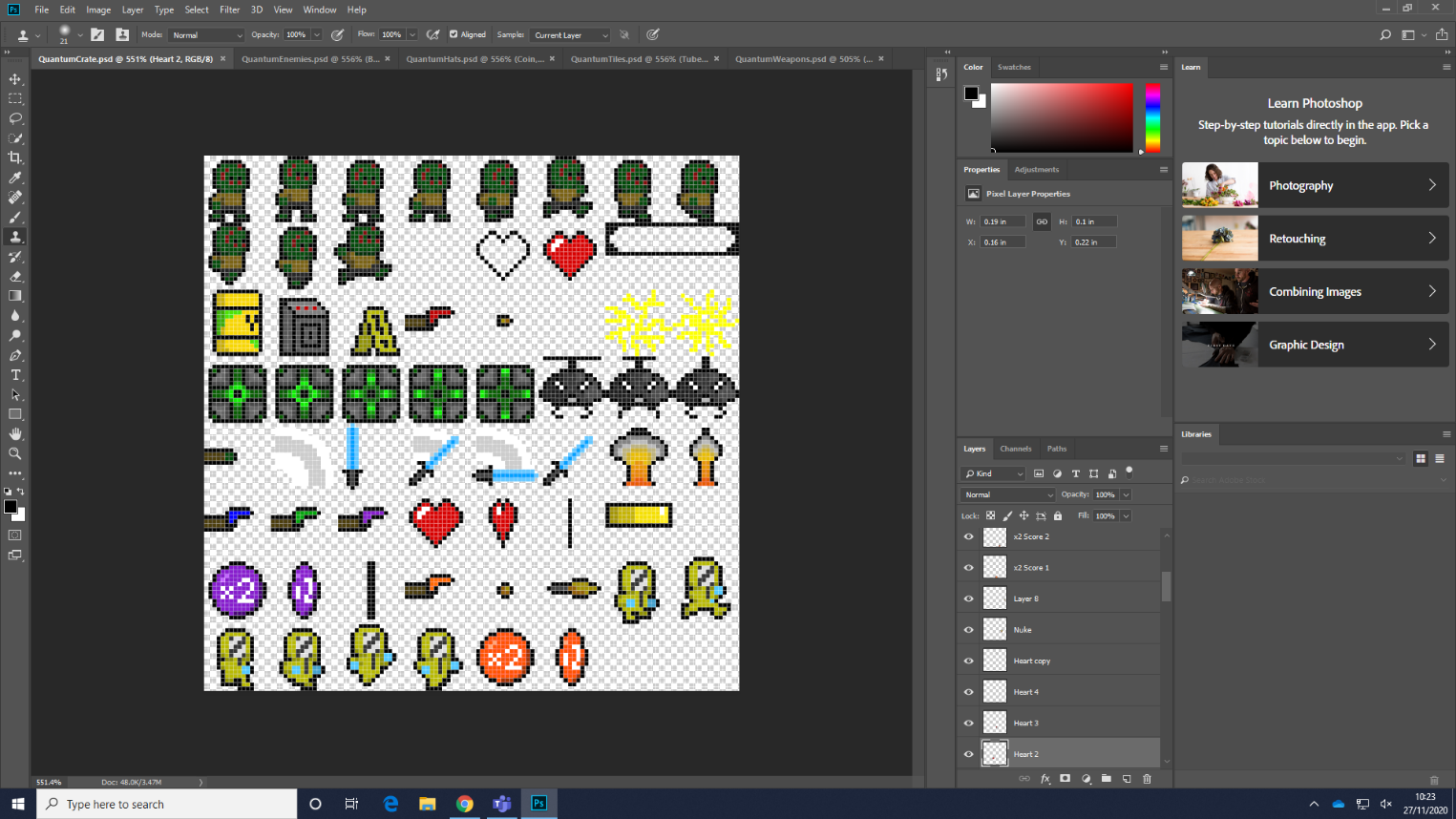
This game is a top-down, local and online PvP shooter available on PC. In order to win, you must kill the other players by any means necessary and be the last standing. Every player begins with a basic weapon, however, throughout each game, more powerful weapons spawn in the arena that the players must try and pick up before their opponents do. A theme throughout the three games is their approach to stopping the player(s) from sitting in a corner and avoiding danger. They use different methods of forcing the player to play using skill and tactics to win. Each game is made up of multiple rounds, which are all the same game, but get slightly different as they progress due to perks. Between every round, each player picks a perk for themselves, which enhances a certain aspect or statistic of their character. For example, speed boost, damage boost, flame effect and health boost among others. These perks add a sense of progression and increased difficulty throughout each game, which is uncommon in multiplayer games.

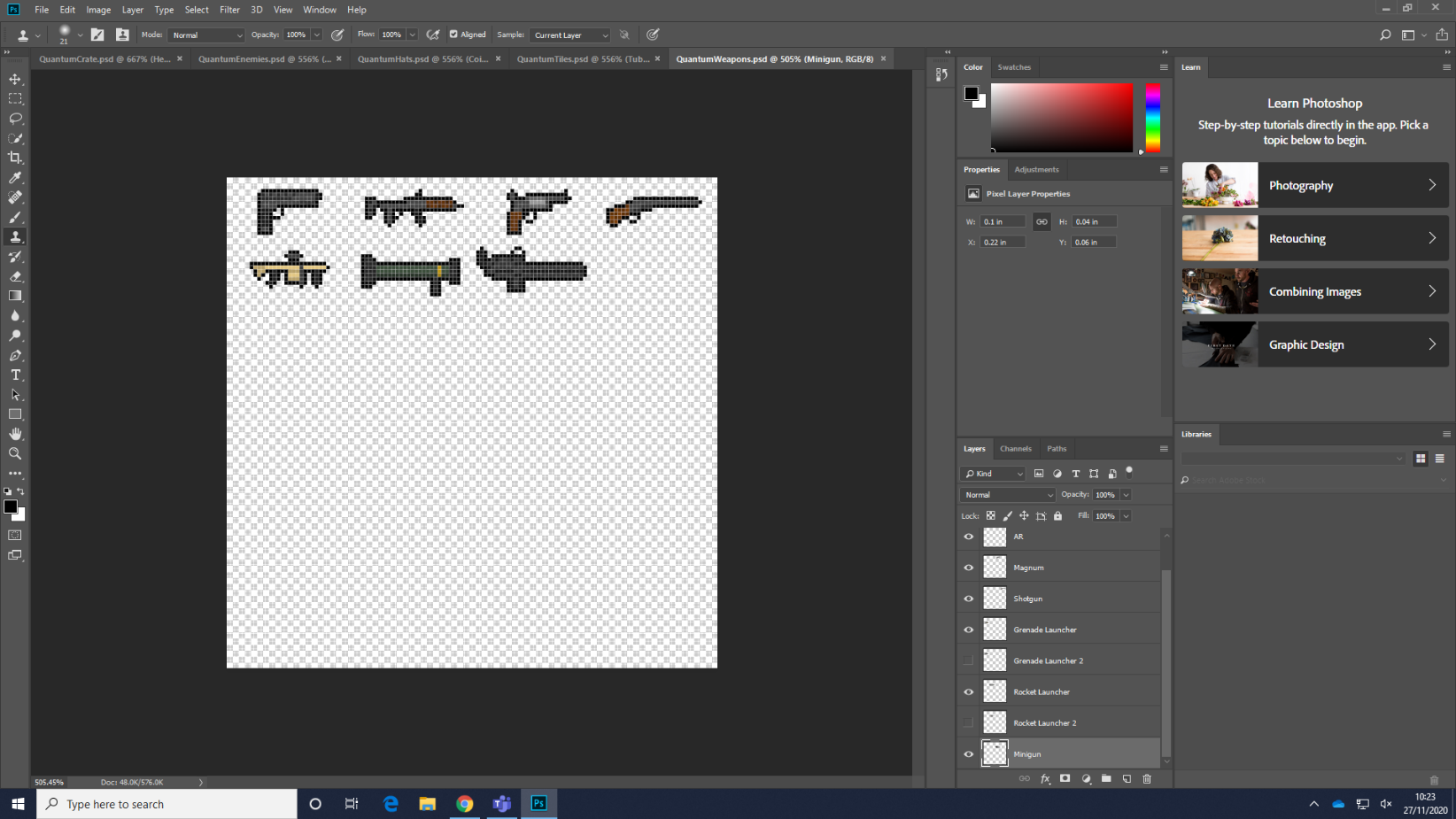
This game is not very similar to the one I am making in style or gameplay but there are a few aspects I could take into consideration. For example, Hyper Jam uses weapons that spawn procedurally, which is what I plan to do. However, I intend to keep the weapon a secret until you have picked it up. This adds an element of luck into the game. I think my game should also have a way of stopping the player from not moving, this could be by using a timer, or perhaps by enemies spawning all the time so they would have to kill them to avoid being overrun. I could also add perks to my game. Whether in the single player section, multiplayer or both, the player(s) could choose perks for themselves as the game goes on.

# A picture containing indoor, sitting, oven, small Description automatically generatedA picture containing table, food, sitting, building Description automatically generated

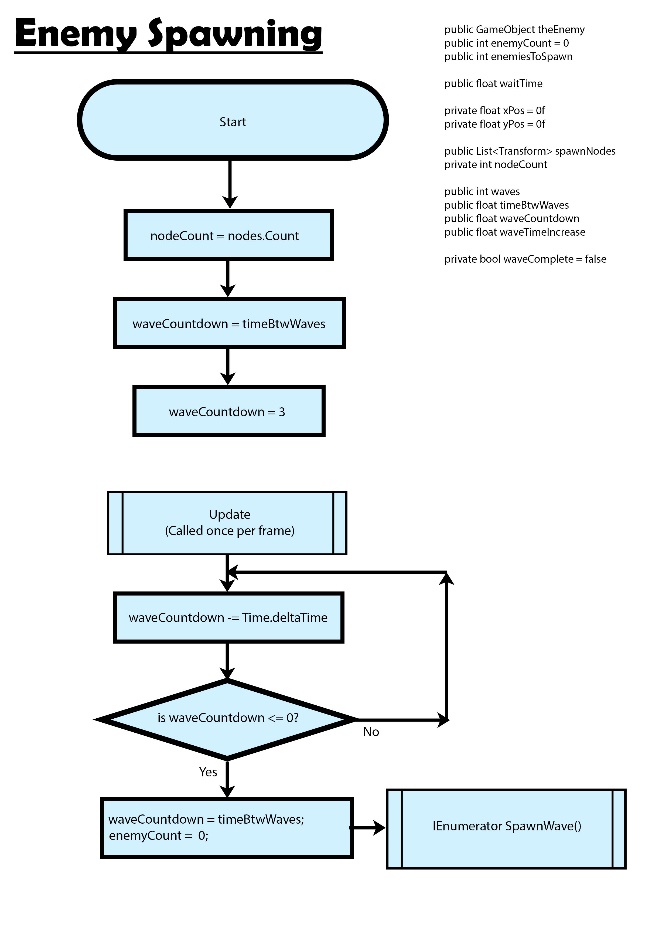
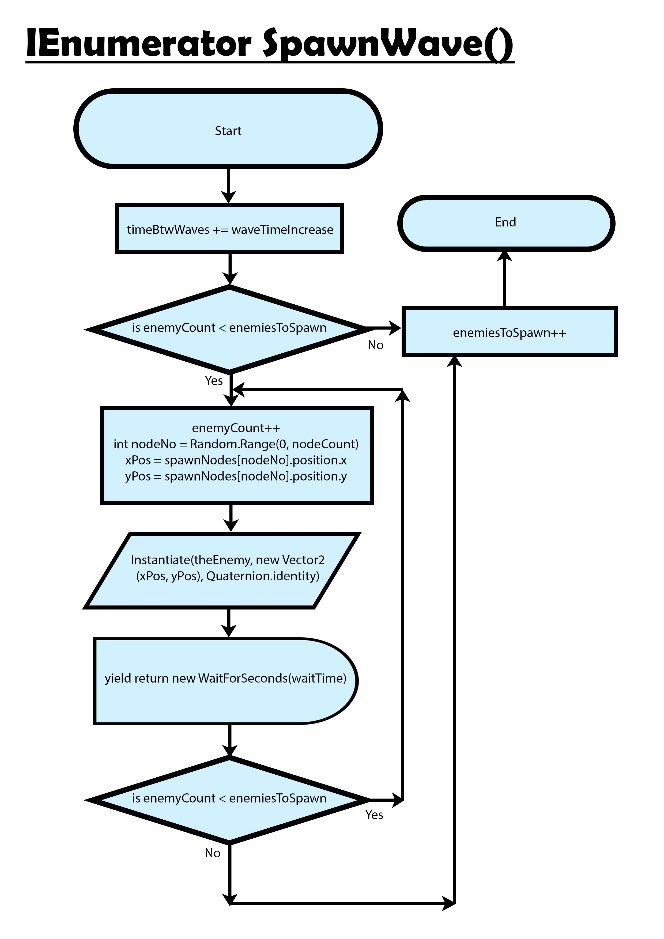
# A picture containing monitor, screen, display, black Description automatically generated

# **SpriteSheets:**





# Problem Modelling



These flowcharts control the spawning of enemies in waves. It begins by initialising the variables for the number of spawn nodes and the countdown to the next wave. In Unity, the update method is called every frame, so it is as if it is running continuously. In the update method, the countdown timer counts down the appropriate amount of time. Then it checks if the timer has reached 0. If it has, it resets the countdown and set the enemy count to 0 so the following procedure can run as if spawning a new wave from 0 enemies. After this, the coroutine SpawnWave() is called. An IEnumerator is a coroutine used to run code simultaneously with any other code in the program. This means the wave can continue to spawn enemies while the countdown timer decreases and potentially even spawns a new wave while the previous wave is still spawning. This should only happen in the much harder levels though.

Each wave will spawn enemies, increase the number of enemies to spawn so it gets harder each wave but also increase the time between each wave so there is enough time for all the enemies to spawn. The coroutine starts by increasing the time between waves. It then checks that the number of enemies that have spawned is less than the number of enemies that have to spawn. This will always be true to start, as we began by setting the enemy count to 0. Therefore, it adds 1 to the enemy count and then chooses a random node from a list at which to spawn a new enemy. Then it creates the enemy game object at the chosen x and y values with a regular rotation (Quaternion.Identity). At the end of this while loop, it waits the given time before spawning the next enemy. I think there should be a small gap between each enemy spawning just so the player is not overwhelmed by every enemy spawning at once. As this functions as a while loop, there is another check to see if the correct number of enemies have spawned. If not, it iterates through again. If so, the number of enemies that are going to spawn is increased by one for the next wave, and the coroutine ends, ready to be called again when the countdown timer reaches 0 again.

# A screenshot of a computer Description automatically generatedA circuit board Description automatically generated

# A close up of text on a white background Description automatically generatedA close up of text on a whiteboard Description automatically generated

These flowcharts demonstrate how the weapon system will work in the game. The algorithm begins by instantiating the variable NoOfWeapons by counting the number of items in the list WeaponList. It also sets both currentWeapon and previousWeapon to “Handgun” as that is the weapon the player begins with each time. Therefore, the stats for the weapon are also set to the stats of the handgun. Each frame, the update method is called. As this happens, the mouse position on the screen is found and set to the variable mousePos. Also, the variable shootTimer counts down to the time when the player can next shoot. As well as this, the algorithm also checks to see if the player has pressed the shoot key. If the shot timer has reached the time of the shot cooldown and the player presses the shoot key, the Shoot() function is called. If either of these criteria are missed, the update function continues. This section makes the arm of the player point toward the mouse. The variable lookDirection is the vector of the player to the cursor, which is found by subtracting the position of the player from the position of the cursor. In this case, the Atan2 function uses trigonometry to find the angle in radians between the x-axis and vector from 0 to the x and y coordinates of the mouse position. Rad2Deg is then used to convert this value to degrees, to make it easier to work with. All that is left to do after this is set the value of this angle to the rigidbody of the player’s arm, so it points in the right direction.

The shoot function starts by setting the shoot timer to 0, as the player is shooting and resetting the cooldown. Then, the bullet is created at the position of the fire point, which is situated at the end of the weapon, with the same rotation of the gun (the gun is pointing in the same direction as the player’s arm as the game objects are linked). Finally, a force is added to the rigidbody of the bullet to make it fire.

Wave 2

Wave 1

# A circuit board Description automatically generatedA close up of text on a white background Description automatically generated

Weapon crates can spawn into the arena using very simple code, utilising spawn nodes similar to the enemy spawning algorithm. However, another way of doing this is to completely randomise the position of spawn, rather than randomising from a set of possible nodes. This would be more interesting for the player as the crates won’t be spawning in the same place every time. When the player touches a crate, the NewWeapon() procedure is called. At this point, a random number is chosen between 0 and the number of weapons in the list. In the next section, the player’s current weapon is removed from the list, making it impossible for the player to pick up the weapon they already have, allowing for more variety in the game. This is why the max value for the random number is noOfWeapons-1. After the current weapon is removed, a new weapon is chosen and the previous weapon is added back into the list, so it can be found in the next crate. The algorithm then uses the weapon’s name to set all the statistics of the weapons. I have only shown 3 so far as examples but more will be added to the game and more variables will be added as more complex weapons appear.

# 

START

nodeCount ← LEN(nodes)

timeToNextSpawn ← waitTime

# Update is called every frame

WHILE Update

#In the update method, elapsed time would be subtracted each frame.

# For simplicity, I am subtracting one second with each iteration.

timeToNextSpawn ← timeToNextSpawn – 1

IF (timeToNextSpawn = 0) THEN

timeToNextSpawn ← waitTime

SpawnCrate(nodeCount)

ENDIF

ENDWHILE

SUBROUTINE

SpawnCrate(noOfNodes)

nodeNo = RANDOM\_INT(0, noOfNodes-1)

spawnNode= nodes[nodeNo]

#At this point, Unity would spawn a crate GameObject at the position of the node

OUTPUT(“Spawn crate at “ + spawnNode)

ENDSUBROUTINE

Trace table using:

Nodes = [node1, node2, node3, node4, node5]

WaitTime = 3

Two iterations of update

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| nodeCount | waitTime | timeToNextSpawn | noOfNodes | nodeNo | spawnNode | OUTPUT |
| 5 | 3 | 3 |  |  |  |  |
|  |  | 2 |  |  |  |  |
|  |  | 1 |  |  |  |  |
|  |  | 0 |  |  |  |  |
|  |  | 3 |  |  |  |  |
|  |  |  | 5 | Eg. 3 | node4 | Spawn crate at node4 |
|  |  | 2 |  |  |  |  |
|  |  | 1 |  |  |  |  |
|  |  | 0 |  |  |  |  |
|  |  | 3 |  |  |  |  |
|  |  |  |  | Eg. 1 | node2 | Spawn crate at node2 |

# 

When the player finishes a game, their score and initials will be saved into PlayerPrefs strings named “PrefScores” and “PrefNames” respectively. In each case, each entry will be followed by an asterisk so they can be easily separated into lists by this algorithm.

A section of code will come before this and get the string of scores and split them into a list of integer scores, by splitting the string every asterisk, then converting each individual score (which will be a string of numbers) into integers. It will then do the same thing with the initials but without converting each item into an integer.

These lists can then be used to display the scores and initials on the leaderboard.

FOR i ß 1 TO LEN(scores)

maxValue ß scores[0]

maxName ß names[0]

FOR n ß 1 TO LEN(scores)

IF scores[n] > maxValue THEN

maxValue ß scores[n]

maxName ß names[n]

ENDIF

ENDFOR

OUTPUT ‘High score:’ maxName + maxValue

FOR x TO LEN(scores) - 1

IF maxValue = scores[x] THEN

Scores[x] ß 0

ENDIF

ENDFOR

ENDFOR

Trace table using:

scores[] = [5, 20, 35, 15]

names[] = [JON, BOB, SAM, IAN]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | maxValue | maxName | n | Scores[n] | Names[n] | OUTPUT |
| 0 | 5 | JON | 0 | 5 |  |  |
|  |  |  |  |  |  |  |
|  |  |  | 1 | 20 | BOB |  |
|  | 20 | BOB |  |  |  |  |
|  |  |  | 2 | 35 | SAM |  |
|  | 35 | SAM |  |  |  |  |
|  |  |  | 3 | 15 |  |  |
|  |  |  |  |  |  | High score: SAM 35 |
| 1 | 5 | JON | 0 | 5 |  |  |
|  |  |  | 1 | 20 | BOB |  |
|  | 20 | BOB |  |  |  |  |
|  |  |  | 2 | 0 |  |  |
|  |  |  | 3 | 15 |  |  |
|  |  |  |  |  |  | High score: BOB 20 |
| 2 | 5 | JON | 0 | 5 | JON |  |
|  |  |  | 1 | 0 |  |  |
|  |  |  | 2 | 0 |  |  |
|  |  |  | 3 | 15 | IAN |  |
|  | 15 | IAN |  |  |  |  |
|  |  |  |  |  |  | High score: IAN 15 |
| 3 | 5 | JON | 0 | 5 | JON |  |
|  |  |  | 1 | 0 |  |  |
|  |  |  | 2 | 0 |  |  |
|  |  |  | 3 | 0 |  |  |
|  |  |  |  |  |  | High score: JON 5 |

# Data Structures

**GenerateGrid:**

2DArray Vector2[,] vertices

This array contains x and y coordinates for every node in the scene. I have variables xSize and ySize to maintain the correct number of nodes for each scene.

**HatController:**

I have used a set of arrays in this script to control the hats that the player wears. I have tested the dimensions for each hat to be placed perfectly on the player’s head, as the sprites didn’t all line up correctly. Each hat has an index number, which is used with each array to get the correct name, size, x translation and y translation.

Array string[] hatNames

This array contains the string names for each hat. When the player buys or equips a hat, the name is found in this array and the index of the name can be used in the other arrays to position it correctly. Eg. “TopHat”, “Fez”, “Fedora”.

Array float[] sizeMultipliers

As the hat sprites are all 16x16 pixelssome needed to made bigger or smaller to look correct on the character’s head. Therefore, each hat has a specific size multiplier to make it look normal. Eg. 0.8f, 0.6f, 0.9f.

Array float[] xTranslations, Array float[] yTranslations

These two arrays position the hat correctly on the character’s head. This is because some hats need to be higher up or further to one side than others. Eg. -0.1f, 0.6f, 0.2f

**HatShop:**

This script is similar to the HatController script as it uses a set of arrays to be applied to each hat with a certain index number each, but this is used when buying and equipping hats.

Array string[] allHats

This array is the same as the hatNames array from HatController. But it controls the hats ina different scene for different purposes.

Array string[] unlockedHats

This array contains all the hats that are unlocked. It works by splitting the PlayerPrefs string of hats the player has bought into an array of strings. The program can then check which hats to display as locked or unlocked based on if they are in the array or not.

Array int[] hatCosts

Each hat also has a cost at its index number. This cost is used when the player decides to buy a hat. They click a hat to highlight it so that the program knows the index number of the highlighted hat. Then, when the player clicks ‘buy’, the cost is deducted from the player’s total money. Eg. 15, 75, 20.

Array bool[] locked

Using the index number again, this array determines whether a hat has been unlocked by assigning a boolean value to each index number.

**HighScoreTable:**

Array int[] scores

This array stores all the scores from previous players who have played the game. It uses a PlayerPrefs string of scores and converts it into an array of integer values. These values are then ordered and placed onto the leaderboard in order. Eg. 30, 240, 95.

Array string[] names

Similar to the scores array, the names array uses a PlayerPrefs string of names the player has entered at the end of each game, separating it into an array of strings. The player is only given three initials for their names. When the scores are ordered, the index of each score is used to order the array of names in the same way. Eg. “LEW”, “LIL”, “MIC”.

**NewWeaponUnlock:**

Array int[] requiredScores

This array is never edited and always keeps the same values. It contains the number of points the player needs to score to unlock the next weapon. It is simply used when the program checks if the player’s total points exceeds any of the values. Eg. 500, 1200, 2500

Array string[] weaponNames

The weaponNames array stores the names of each weapon with index positions corresponding to the scores required to unlock each weapon. If the program finds that the player had reached one of the scores and they haven’t already unlocked that weapon, that weapon is added to the list of available weapons. Eg. “Shotgun”, Minigun”, “Grenade Launcher”.

**Pickup:**

Array string[] pickupNames

This array contains each pickup that is available to the player. When a pickup spawns, a random one from the array is chosen. Eg. “Extra Life”, “X2 Coin”, “X2 Score”.

**WeaponController:**

This is one of the largest and most important scripts in my program. It contains lots of arrays, which use specific index numbers for each weapon.

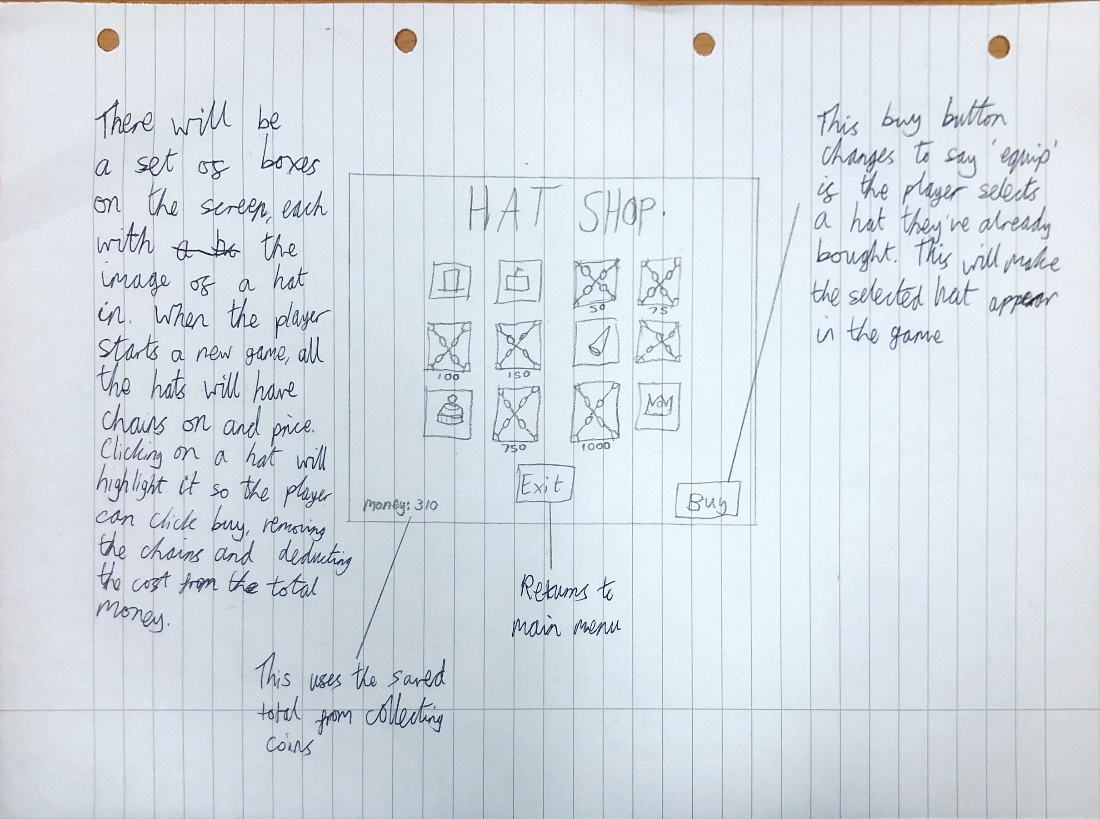
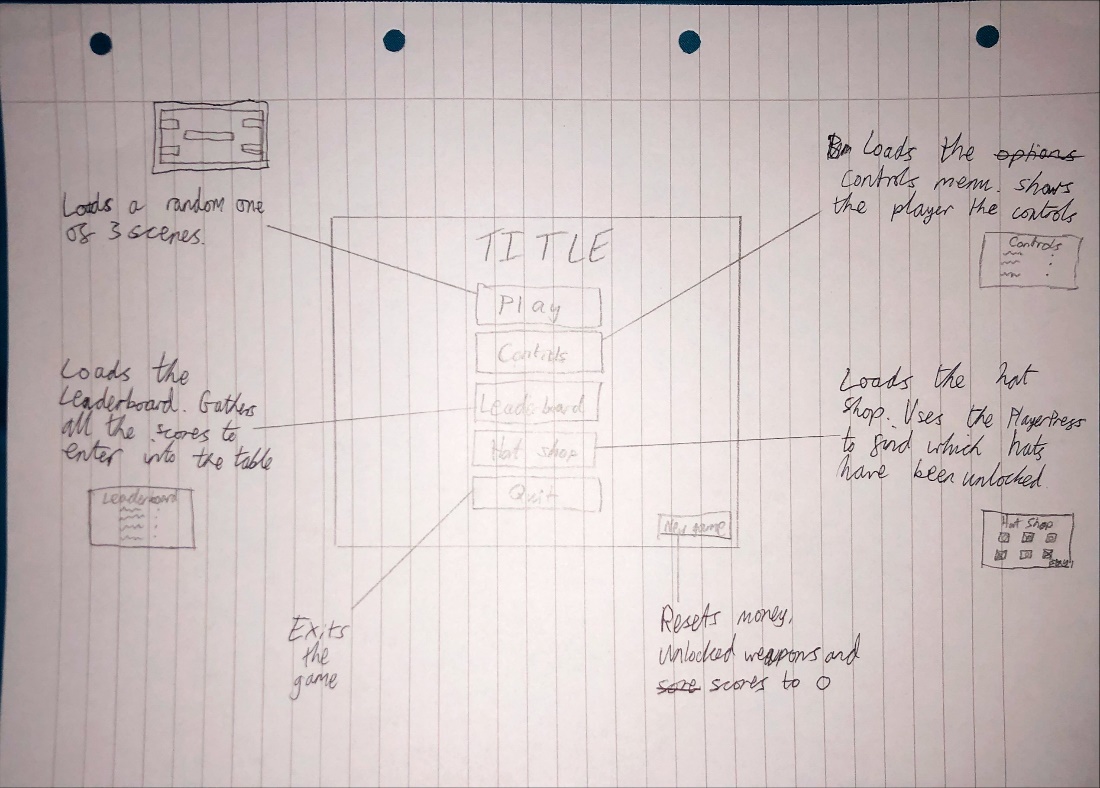
Array string[] weaponList, Array string[] allWeaponList

The array weaponList contains the weapons the player has unlocked, while allWeaponList contains every weapon that the player has unlocked and can unlock. These are used to find the index position of a weapon to be used in all the other arrays. Eg. “Handgun”, “Assault Rifle”, “Magnum”.

Array int[] WeaponBaseDamages, Array int[] bulletCounts, int[] bulletForces, Array float[] shotCooldowns, Array float[] buletSpreads

These arrays all contain the stats for each weapon. When a new weapon is picked up, the program gets the index number for that weapon. It then sets all the stats of the weapon to the ones at its specific index position.

My game will be an arcade-style game, assembled as if it is going to be played by many different people, yet entirely supports only one player. This style is achieved by the fact that the player can play the same type of game (with various different layouts) and aim to get the highest score possible before dying, at which point they will be asked to enter their initials to be stored in a scoreboard before trying again. The scores and initials will each be stored as separate PlayerPrefs values. Then, when the scoreboard is loaded, these PlayerPrefs will be sorted into lists, which can then be outputted into the scoreboard. The highest score will be found, and the initials that correspond to this score will also be found using the index position of the score. Both will then show up in the first position on the scoreboard, then both will be deleted from their lists. However, they are not deleted from storage, only the temporary lists, so they can still be accessed the next time a player loads the scoreboard. This happens 10 times to fill all 10 positions on the scoreboard. When the player starts a new game, the scoreboard will be filled with predetermined scores so the player can see it as a sort of goal to beat them all progressively.



# Research

**A\* vs Djikstra’s**

Both of these are graph traversal algorithms. They can be used in games as pathfinding to move an enemy toward the player or any other object in the level or scene. I am planning on using one of these in my game to move the enemy towards the player in order to melee attack. Both algorithms have their merits, but I need to use the one that is most suited to the code style, engine and the game itself.

**Djikstra’s**

Djikstra’s algorithm was created by Edsger W. Dijkstra in 1956 to determine the shortest path between two nodes. It was originally used to find the shortest distance around the roads of Amsterdam. It was also utilised by hardware engineers to work out the shortest length of wire needed connect pins on the back of a machine. This algorithm is praised for its versatility as it is able to be used in many different scenarios and can solve a wide variety of problems.

Djikstra’s algorithm works by setting the weight of the paths to untraversed nodes to infinity. This ensures they cannot be considered as the shortest path until they have been explored.

**A\***

The A\* algorithm is used frequently in computing due to its optimal efficiency. Unlike Djikstra’s algorithm, A\* was invented for a computing purpose, rather than a real-world scenario adapted to solve computer-based problems. It was created to help manufacture a robot that could plan its own movement and was used at this point simply to find the path with the lowest weight/cost, given that it is the sum of all the edges within the path. However, the A\* algorithm has proven to be useful in many situations, mostly used in computing.

The A\* algorithm employs a *heuristic estimate*, which decides on an estimate to the most efficient route from one node to another. The algorithm often uses the terminology *g(x) + h(x)*, where *g(x)* is the exact weight of the path from the starting node to the node *n* and *h(x)* is the heuristic estimate of the distance between node *n* and the destination node. The A\* algorithm uses successful elements from Djikstra’s algorithm that favours the nodes closest to the starting node (*h(x)*) as well as favouring the nodes closest to the destination node, similar to the best-first search algorithm (*g(x)*).

**Which is Better?**

Djikstra’s algorithm is a form of the A\* algorithm when *h(x)* = 0 for all nodes. One major drawback of Djikstra’s algorithm is the fact that it has no heuristic. Because of this, it searches in every direction, while A\* only searches in the direction of the destination node, making it much faster. Due to this fact, Djikstra’s algorithm can be seen as a less efficient version of an A\* algorithm. However, Djikstra’s algorithm has an order of n2 so can be used for relatively large graphs or models. Additionally, A\* can only be used when searching for a known destination, while Djikstra’s can be used to search for the closest of many possible destinations. Also, Djikstra’s algorithm is in general, easier to program.

**Conclusion**

After having researched both forms of pathfinding, I have come to the conclusion that A\* is faster and the most optimal form of graph traversal but can only be used when the start and end nodes are known. On the other hand, Djikstra’s algorithm is versatile so can be used in many scenarios and solve a wide variety of problems and it is easier to implement. However, it is drastically less efficient than A\*.

In my game, the starting node and the destination node will both be known, as they are the enemy and the player, respectively. Therefore, I feel as if it is most appropriate to use the A\* graph traversal algorithm to control the enemies’ pathfinding.

**Collision Detection**

Unity offers its own forms of collision detection through the use of colliders of various different shapes. As I am making a 2D game, I could use colliders that are square or circle shaped to allow the players and enemies to collide with objects in the level, as well as each other. Unity also has a built-in procedure called OnTriggerEnter2D, which detects when the collider on an object collides with another object. I could then use this procedure to then decide if an enemy should take damage, or a player should get a new weapon, or the score should increase or any number of things.

One problem with these kinds of the colliders is the lack of accuracy. For example, I could use a circle collider for an enemy, but they might not be entirely circular. This could be confusing for the player as they may not interact with the enemy in the way the sprite would imply. One way to fix this would be to use pixel perfect collision detection. This is a complex algorithm that uses the sprite data to create a custom collider that perfectly covers all the pixels in the sprite, but no others. This algorithm seems to be extremely useful because, not only will it allow enemies to have colliders that are perfectly covering their bodies, but the collider should also update with the animation. Additionally, the pixel perfect collision detection should work with everything, not just the enemies. This includes the player, items in the levels and even the terrain of each level. However, some objects might not need this complex of a collision detection system, if it stationary with a simple sprite.

**Decision Trees**

**ID3**

ID3 stands for Iterative Dichotomiser 3. It is an algorithm used to generate a decision tree from a set of data. It works by assigning the original set S to the root node. With each iteration, the algorithm checks every unused attribute of the set S in order to calculate the entropy - H(S) - and information gain - IG(S). Entropy, put simply, is the amount of surprise that can possibly occur within a certain event. Information gain is the overall change in information entropy from one state to another. The algorithm then selects the attribute that has the smallest entropy or largest information gain and partitions the set S based on the selected attribute. It uses a greedy strategy, meaning it converges on the local optimum in each section. This means that the algorithm does not guarantee an optimal solution, as it is only finding the best solution at each individual section. One way that it can be improved is by introducing backtracking, which allows the node choices to be made more optimal after they have already been selected. Although this improves optimality, the algorithm could possibly take longer to run. Another issue with ID3 is its tendency to overfit the data. This means the algorithm contains more parameters than can be justified by the data. Luckily, this problem can be easily averted if the user sticks to using smaller decision trees.

**FSM**

FSM stands for Finite-State Machine. It is a mathematical model of computation that can be in one of a finite number of states at any given time. The FSM can react to inputs and change states; this is called a transition. An FSM generally consists of a list of states and a list of inputs that can trigger transitions. As well as this, the initial state may also be stored as a starting point for the algorithm. This is a simple style of algorithm as it just determines what a program (or a specific object in a program) is doing at any given time. The status of an object or system is known as a state and is waiting to execute a transition to another state. A transition is simply a set of actions that are executed when a certain condition is fulfilled; this condition could be a user input, an incrementing variable reaching a certain value or many other things.

**Which is Better?**

Overall, I think a finite state machine would be better suited to my program as ID3 can often add unnecessary complexity to a program. As well as this, FSM is much simpler, and I think as the decision tree will only be attributed to a single character, there’s no need use an in-depth ID3.

# Objectives

1. Player character will be controlled using WASD to move and space to jump.
2. Player character will be able to shoot by aiming and clicking with the mouse.
   1. The player’s weapon will always point in the direction of the mouse position.
   2. When the player shoots, the bullet will fire in the direction of the mouse position.
      1. The bullet makes a particle effect and is destroyed when it hits something.
3. Enemies spawn into the level.
   1. Enemies will spawn into the game in waves. Each wave will spawn after a set length of time. With each wave, the number of enemies spawning and the time to complete the wave will increase.
   2. The spawn point of each enemy is chosen at random from a set of spawn nodes.
   3. If the player shoots an enemy, that enemy loses health
      1. When the enemy has 0 health, it will die, explode and the player will get points.
   4. There will be at least 3 different kinds of enemies that can spawn.
      1. Some enemies could attack the player.
      2. Some enemies shoot the player.
      3. Enemies damage the player by colliding with them
   5. Enemies take away health from the player.
   6. The regular enemy moves toward the player using A\* pathfinding.
4. Crates spawn into the level periodically from a random point at the top of the arena. They fall and stop when they hit the ground.
   1. The player can pick up the crates by colliding with them using pixel-based collision
      1. A particle effect will appear when a crate is opened.
   2. The crates will only be affected by a player touching them. Enemies pass straight through them.
5. The player picks up a new weapon when they open a crate.
   1. This weapon is chosen at random from a list of weapons.
   2. The name of the weapon appears on the screen.
   3. The player will then have access to all the attributes of the new weapon (E.g. Assault rifle fires very quickly).
6. The player can press a key to open the pause menu.
   1. The pause menu will suspend the game, disabling player and enemy movement.
   2. The player will have options to choose from in the pause menu.
      1. The player will be able to return to the game.
      2. The player will be able to view the controls.
      3. The player will be able to restart the level.
      4. The player will be able to return to the main menu.
7. The player will be able to change their character’s appearance
   1. This could be done from the pause menu or a completely separate area from the main menu.
   2. The player will unlock new outfits to customise their character with. This can be done by either buying them with money or progressing in the game.
8. The player will unlock new weapons as they play the game.
   1. A new weapon will be unlocked when the player reaches a certain point or accomplishes a certain task in the game.
   2. The player will be told they have unlocked a new weapon.
   3. The new weapon will be added to the list of possible weapons so it can be chosen at random the next time the player opens a crate.
9. The player dies when they reach 0 health.
   1. A particle effect appears when the player dies.
   2. The player loses a life.
   3. If the player has no lives left, they are given the option to restart or return to the main menu.
      1. If the player restarts the level, the scene is reloaded.
      2. If the player chooses the other option to return to the main menu, the main menu scene is loaded.
10. At certain points in the game, a boss wave will begin.
    1. Instead of spawning many enemies for the player to fight, the boss wave will consist of one stronger enemy.
       1. This enemy will shoot projectiles in the direction of the player.
       2. The boss will move side to side at the top of the screen, following the player.
       3. When the boss’ health goes below a certain point, he will heal himself.
       4. The boss could spawn some enemies while he shoots at the player.
       5. The boss could also have an attack that fires lasers at the player. These lasers would appear at random places on the screen, possibly based on where the player is at the time.
       6. The boss’ actions will be controlled by a finite state machine.
       7. The boss could enter certain states in response to the player’s actions, for example where they are on the screen. Aside from that, a random state will be chosen.
    2. When the player kills the boss, they will get a lot of points and coins.
    3. The boss will spawn at a specific wave each game.

# Limitations

As I am using Unity, a very well-made and technologically advanced game engine, there are not many technical limitations concerning my game. However, one thing I decided would be too complicated to implement is online play. There is an entirely separate section of Unity that handles multiplayer games on separate machines. Not only is this complicated, but it would also be near impossible to test. I only have one computer so it would be hard to work out if the online section works and it would be even harder to fix it.

After having considered the use of Unity further, I have realised how it restricts my ability to implement my own complex algorithms. Unity is very user-friendly and contains many functions that are available for the user to make their game better, such as pathfinding and collision detection. If I am going to use Unity, I will have to create my own versions of many of the algorithms Unity provides for the user in order to demonstrate a comprehensive understanding of programming.

At some point, I may be able to add a story mode into the game. I will start with an arcade mode where the player can play the regular game with lots of enemies spawning and the game gets harder. This mode will be endless so the player can continue progressing as long as they keep getting better at the game. A story mode is more complicated as I would have to design every level and determine where and when enemies are going to spawn.

Overall, I think it is unlikely that a story mode will be very effective or even possible. Firstly, the game would be entirely based around the arcade-style system, except for the story. I feel as if I have planned all the features of the game around the version where the player tries to get the highest score. I don’t feel as if a story mode would fit very well with this, and I am unsure as to how the two modes could be related. Equally, there needs to be a clear difference between the two modes, aside from the button you click in the main menu. Otherwise, players could get confused and be uncertain as to which game mode they are playing.

# Feasibility of potential solutions

I am using C# for my project. C# is a general-purpose, object-oriented programming language developed by Microsoft.

Here are some advantages of C#:

* It is seen as one of the easiest languages to learn and can be used in a wide variety of areas and for many purposes.
* C# builds on the already powerful C++ and adds extra functionality. It also has lambda expressions and direct memory access, giving it superiority over other languages of a similar calibre such as Java.
* C# is an open source software, meaning anyone can download and use it, and anyone can collaborate and add their own extensions and functionality to the language.
* It is cross-platform. A program written in C# can be used on any operating system such as android, iOS and windows.
* C# compiles and executes very quickly compared to other languages.
* C# has a large library. There are a lot of inbuilt functions people can use to make any kind of development quick and easy.

I am also using the Unity game engine. This software allows you to use C# code to control sprites in a 2D or 3D area, known as a ‘scene’.

Here are some advantages of Unity 2D:

* It is very simple to use once you know the basics. It is one of the most user-friendly game engines.
* Because it is so popular, there are many tutorials on the internet on how to use unity at a beginner, amateur and even professional level.
* Unity is very visual, which is helpful to the user. All the sections of the Unity editor are very clear and simple even to a beginner.
* Objects and sprites can be easily placed directly onto a scene and easily altered. This allows the user to quickly made amendments without having to change a lot of code.
* Unity can directly access spritesheets and art files that are saved in the project’s files. When these art files are edited, the sprites in Unity are updated too.