



PROJECT REPORT



Contents

Contents	1
1.0 Analysis	4
1.1 What is it?.....	4
1.2 Why not paper?.....	4
1.3 Target Audience	5
1.4 Similar solutions and ideas:	5
1.4.1 Similar project: Destiny 2.....	5
1.4.2 Similar project: Soul Knight.....	7
1.4.3 Similar project: Team Fortress 2	8
1.4.4 Good Ideas.....	9
1.5 My solution	9
1.5.1 The main ideas	9
1.5.2 Aspirational features	10
1.5.3 Limitations	11
1.6 Software Requirements	11
1.6.1 Game Requirements	11
1.6.2 Database Requirements.....	12
1.7 Hardware Requirements.....	12
1.8 Success criteria checklist	12
2.0 Sprint 1 – Player Movement.....	14
2.1 Design.....	14
2.1.1 Algorithms.....	14
2.1.2 Key Variables/Structures	15
2.1.3 Validation	16
2.1.4 Data to be tested	16
2.2 Development	16
2.2.1 Movement Script	16
2.2.2 Mouselook Script.....	19
2.3 Testing.....	20
2.3.1 Tests	20

2.3.2 Fixes	21
2.4 Sprint Overview & Maintenance.....	22
3.0 Sprint 2 – Level Generation	22
3.1 Design.....	22
3.1.1 Algorithms.....	22
3.1.2 Key Variables/Structures	24
3.1.3 Data to be tested	25
3.2 Development	25
3.2.1 Resources	25
3.2.2 Level Generator	25
3.3 Testing.....	27
3.3.1 Tests	27
3.3.2 Fixes	29
3.4 Sprint Overview & Maintenance.....	29
4.0 Sprint 3 – SQL Server.....	30
4.1 Design.....	30
4.1.1 Algorithms.....	30
4.1.2 Prototyping / Normalisation	32
4.1.3 Validation	33
4.1.4 Key Variables/Structures	34
4.2 Development	35
4.2.1 PHP Scripts	35
4.2.2 Database tables	39
4.3 Testing.....	39
4.3.1 Tests	39
4.3.2 Fixes	40
4.4 Sprint Overview & Maintenance.....	40
5.0 Sprint 4 – Modular Items.....	41
5.1 Design.....	41
5.1.1 “Type Structure”	41
5.1.2 Use functions	42
5.1.3 3D model swaps.....	44

5.1.4 What to test.....	44
5.2 Development	44
5.2.1 Use Functions	44
5.2.2 3D Model Swaps	46
5.3 Testing.....	48
5.3.1 Tests	48
5.3.2 Fixes	49
5.4 Sprint Overview & Maintenance.....	50
6.0 Sprint 5 – Enemies (angry cubes)	51
6.1 Design.....	51
6.1.1 Entity Class	51
6.1.2 Enemy States	52
6.1.3 Key Variables	54
6.1.4 What to test.....	55
6.2 Development.....	55
6.2.1 Entity Script.....	55
6.2.2 Enemy Script	57
6.3 Testing.....	60
6.3.1 Tests	60
6.3.2 Fixes	61
6.4 Sprint Overview & Maintenance.....	62
7.0 Evaluation.....	63
7.1 Final Tests	63
7.2 Maintenance.....	64
7.3 Success Criteria Recap	65
7.4 Summary.....	66
Bibliography	67

1.0 Analysis

1.1 What is it?

An issue with many single-player games, is that you cannot continue playing from multiple machines, and the game can become stale and repetitive because they often have a single goal that once complete, the game cannot provide any more entertainment.

The aim of this game is to try and fix those issues, by making your items stored in your account, allowing you to continue from anywhere, and by removing that single goal, and replacing it with simply trying to obtain the most fun or powerful items. This goal allows players to aim for whatever items they want, letting them develop their own playstyles and attachments to items that keeps them entertained. Furthermore, a random generation system will generate the levels so they can be replayed many times without feeling overly bland or even identical.

The game itself will be a rogue-lite style game, combined with elements of the looter-shooter genre. Rogue-lite elements will come from repeating levels or environments that vary each time using random generation. It will also be a rogue-lite and not a rogue-like, meaning that players will retain some or all items between levels, which allows the looter-shooter aspect of the game to flourish.

1.2 Why not paper?

This game could be considered similar to things such as dungeons and dragons, or other tabletop games, which can be played simply with people and an imagination. The benefit of using a computer system to play this game, means that it can be easily visualized, and it can provide a much faster paced experience than having to imagine and describe everything.

To achieve this, computational methods can be used. Abstraction and divide and conquer allow the project to be broken down into smaller problems, and each of those broken down into individual things that can be scripted or made. Object Oriented Programming (OOP) can also be used to keep parts of the project separate and allow data to not get mixed up between different sections. Finally, heuristics allows more complex problems that seem to have many complicated parts to be simplified, and unnecessary parts can be removed or imitated without actually needing anything complex to be done to them. For example, instead of computing actual bullets with physics, you can just draw a line from an equation and see what is at the end of the line and use that as the object that is hit.

Using a computer system also allows for data to be stored long term, enabling the goal of making the game easy to put down and pick up again whenever and wherever you are.

This will be done using a client-server model where the server provides all the players unique data, and the client just has generic code that can be copied as many times as needed and makes the game playable.

1.3 Target Audience

This game is aimed at people aged between 16 and 20 years old. This is because people this age tend to have free time that needs filling and are also the age group that play video games the most. However, this age group specifically was chosen and not 12 to 20, because people between 16 and 20 years old have a slightly more limited free time. This causes them to look for games that can be played for a long time overall, but with actual play sessions being quite short, with fast rewards and results.

This fits the design of the game, as levels are designed to only be played for as long as you can or want and then ended at any point. Rewards are then given for how far you have gone and what you did. Therefore, players can easily dictate how long they play for and when they play, without feeling the need for time commitments such as in longer story focused games, or competitive online games.

Additionally, people younger than 16 shouldn't play this game as the main way to make progress and earn additional items is through completing levels with scaling combat difficulties. Because of this, people who are younger may struggle with increasing difficulty so could only play for a short time before losing. Secondly, they should also not play it due to the large amount of combat that would take place between the player and enemies along the way which might not be appropriate.

1.4 Similar solutions and ideas:

1.4.1 Similar project: Destiny 2

"Destiny 2" is a looter-shooter game by "Bungie", which means that it primarily focuses on the acquisition of better gear through completing levels. It focuses on being an online game where players make long-term characters that they progress and build a connection to.

Positives And Negatives

Firstly, it successfully holds players' attention by making the progression system feel rewarding. They do this by having a wide variety of different types of gear that players can use, with more powerful options becoming available as the players become more powerful themselves. This creates a game loop where players find their favourite items to use and then define their own playstyle using these items, while also constantly upgrading items and becoming more powerful.

Secondly, the "endgame" of destiny is often seen to be the player's appearance. This is achieved by having an even bigger variety of armour items that are visible on the player

and the system is so unique from other games as all items can be earned either through skill or time spent playing. This causes items to have a greater value to the player, so making their character look good is one of the most important parts of the game.

Finally, the data and player inventories in the game, along with the quests and items have all been stored on an external database that the client interacts with to use and modify different player inventories. This model allows for players to play anywhere using their account login and retain all their progress. Furthermore, it allows the integration of systems such as stat trackers that can allow players to be competitive over things like who looks the coolest, who has the fastest times in challenges, and who is the highest level. For example: [DIM](#) allows players to see each other's items on a website externally to the game (shown right).



However, one downside to the game is that all levels are manually created and there are a finite number of possible things to do. While manually created levels make for higher quality and more interesting levels, in a game where you are expected to play those levels over and over again, they can become repetitive to the point people just stop playing. Because of this, I have chosen to include both manually created areas and randomly generated areas in the project. This is so that environments will have areas that are the same but more interesting and detailed, and also transition areas that will be different each time meaning you can't memorise level layouts.

What can be reused?

The idea of non-linear progression maintains player interest for extended amounts of time, while allowing the player to define their own progress. This can also be quite simply implemented by scaling item drop's stats to the player's overall level, or by locking items behind certain challenges/secrets/quests. Therefore, this should be included as it adds interest and progression to the game with relatively simple systems.

Having many different visual items could be a potential feature as it allows players to feel more connected to their character, however this requires creation of every single one of these unique items. I believe it is out of the scope of the project to create the number of items needed for this level of player-uniqueness, so instead the database will be designed to be scalable, so that there is the possibility for them to be added, given time.

Finally, the design I am planning to use for the item database is very closely modelled around the Destiny 2 database; it mostly focuses on storing numerical stats, names and

IDs of items, and lets the game client use those to create the item in-game. This also allows items to be added easily without modifying the game much; you would only have to upload a 3D model into the game assets and add its data into the list of existing items in the database and it should be available in-game.

1.4.2 Similar project: Soul Knight

“Soul Knight” is a mobile rogue-lite game by “Chillyroom”, meaning that it focuses on a repeatable structure where you play through the same levels to make your character more and more powerful.

Positives And Negatives



Unlike Destiny, this game doesn't have handmade levels for the players to play, but randomly generated ones. This can make the game different every time and allow the player to play for much longer without becoming bored. However, this could also be seen as a downside as levels end up being relatively similar to each other, and often not very detailed.

Additionally, the way weapons are obtained in this game is pure random chance and luck. To obtain a weapon you simply play over and over until it drops randomly from a chest or enemy. You then must repeat this 5 times to unlock the weapon for yourself. This could be a good thing because it allows any player at any skill level to obtain everything however it could also become extremely tedious. This is because all items can drop from everything, so if you want one specific item, the chance of getting it is miniscule.

Finally, any items you obtain aren't actually kept, simply discarded at the end of every run. This can cause the items to seem meaningless and doesn't take advantage of the fact that humans get attached to objects very easily. Instead, they must be re-bought before every level, and only if you have unlocked them, which can cause progression to seem non-existent as even when you unlock stuff you can't use it unless you buy it.

What can be reused?

The random generation will most likely be reused as it allows a small amount of assets to be reused for every level and will prevent lots of time being wasted on making levels. However, as this can cause boredom from low-detail levels in “Soul Knight”, the random generation should combine premade and random levels by fitting together premade pieces according to rules, similar to a jigsaw.

The way weapons are obtained is extremely slow and tedious and there is no variation or control over what you obtain. While the point of this game is to make the player play levels repeatedly for items, there should still be some level of control over what is

obtained. Instead of being pure randomness, there should be ways to control it, for example quests that reward items, or certain enemies and chests being more likely to drop specific items.

Finally, the fact that items are lost every run in “Soul Knight” takes away a large amount of the sense of progression, so in this project, obtained items should be permanent and allowed to be used as soon as you find one.

1.4.3 Similar project: Team Fortress 2

Positives And Negatives

“Team Fortress 2” is a first-person shooter game by “Valve”, meaning that it focuses on gunplay and many different weapon variations. It also focuses on combat between players using these weapons.

As the primary focus of this game is on shooting, lots of concepts and methods for implementing these guns can be seen in it. For example, in TF2 the bullets are handled by the game in one of 3 ways.

Firstly, the game uses a method called “hitscan” to determine where bullets from weapons land and what they should affect. This means that there is no actual bullet, but the gun itself simply checks what object is directly in front of it when it fires, and the world space coordinates of where the bullet hits. It does this using a raycast which allows it to then do actions on the item that was “hit”. These can range from creating impact marks, or damaging the object hit or calling functions on that item.

Secondly, TF2 also uses projectiles for some weapons such as rockets. These work differently to “hitscan” weapons in that they create a game object that is treated as a bullet. This then travels forwards in the direction the gun was facing, like the ray used in “hitscan”, except it takes time to travel that distance. If it hits anything on the way that is the object that will be referenced to do any of the actions that hitscan could do.

Finally, it also uses “physics projectiles” that work similarly to a normal projectile in that there is a game object except this time it has physics applied to it. This means it could bounce off surfaces or interact with other physics-based objects, while still affecting any hit objects with the same methods as the other two types of firing.

What can be reused?

Overall, using the three different types of bullets would be a relatively simple and effective way to introduce variation to different types of weapons, making them feel different to use, and to have different techniques for using them. However, I will most likely only implement “hitscan” and “projectile” weapons, as physics-based weapons would be far more complicated. Because of this they could be an aspiration for the project to include.

1.4.4 Good Ideas

- Item drop level scaling with player level (D2).
- The client constructs items from database information/stats (D2).
- Scalable database so more items can be added (D2).
- Tile-Based random generation (SK).
- Some, but not full, randomness to item drops (SK).
- Three types of weapon shooting calculations: Physics, Projectile, Hitscan (TF2).

1.5 My solution

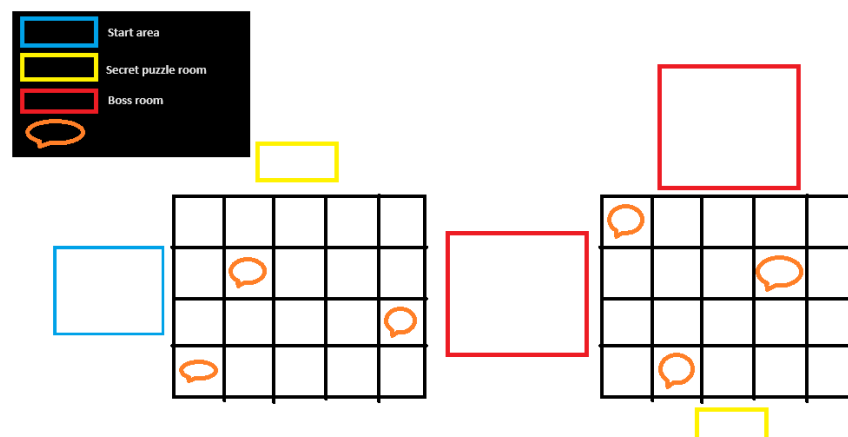
1.5.1 The main ideas

The project will focus around two main aspects: random 3D level generation using the Unity game engine and storing data online to allow progress to be saved wherever you play the game using SQL and php.

Random Level Generation

Each level will consist of premade areas, chosen and placed randomly, and the spaces between them will be filled with a randomly generated grid of rooms with small challenges and enemies in them.

The premade rooms can be things like the spawn area, secret puzzle rooms, boss fights, and if story quests are added, rooms relating to those.



The random generation will work by going through the grid of spaces and comparing the surrounding tiles. Based on the surrounding tiles it will identify which tile it can generate and will choose a random one of those. For example: if a space is on the edge of the grid, it must have no openings on that edge, however if it is against another open space, it can have different openings in it that can link to other adjacent rooms.

The reason for random level generation being an essential feature to the project, is because it aligns with the project goal of making the game entertaining for extended number of times. It does this by making levels replayable and allowing the player to find different objectives, layouts, and items each time.

Data Persistence and Storage

The player's data will be stored under their username in an SQL database, allowing them to play from any machine with the game and internet, as long as they keep their login. This also allows progress to be externally tracked, making adding features such as achievements, or sharing progress with friends easier.

It will also contain definitions for all the obtainable items in the game, so that they can remain consistent when different people obtain the same item. This allows for people to want specific items and gives people a goal to work towards as well as getting as far through the levels as fast as possible.

Items will be stored using 3 key values: name, type, and stats. The name is simply a name for people to identify it with, while the type defines what sort of item it actually is and allows it to use different functions within the game. The stats are a set of numeric values that can be tuned for each item to make them behave differently. For example: fire rate for guns, amount of hp recovered for consumables, or the swing speed of a sword.

Storing the items like this allows the same template and tables to be used for all items, while the items still remain unique from each other.

This is also an essential feature to the game as it enables the project's other goal; to allow the player to play whenever and wherever they want and for as long or short a time as they want. It does this by making their data accessible from anywhere, no matter where or when they stopped playing.

1.5.2 Aspirational features

The game could also include a quests system, for example having characters in the spawn area, or hidden throughout the levels, which give you objectives. These could possibly give you extra and rarer rewards, or could be used to tell a storyline, or introduce new items or areas into the game. This would be fairly simple to implement, but creating smaller stories for quests could be a longer and much more in-depth task given time.

It could also potentially include a local multiplayer game mode, where two friends both log into their accounts on the same computer, and both either play together in co-op, or play against each other to see who is more skilled or has found the best items. This would be harder to implement however, as multiple players would have to be handled at once without their data mixing or getting changed incorrectly.

Finally, a feature that would be more immersive and make the player feel more personalised, would be making the armour items the player earns show on their character. This would allow players to all look exactly how they wanted, and also give them another reason to keep playing to obtain the looks they wanted. However, this

could be difficult to implement well, as it would mean making and animating a character, and giving it the functionality of having armour stay in position during its animations.

1.5.3 Limitations

As the random generation is only going to work on the levels, any items that are dropped must be made manually. This can be good as it should prevent any too powerful or broken items from being made, but it is also a large amount of time to simply create items by tweaking their values and creating 3D models. This also means that if the main goal of the game is to try to get the best items, this can only be a substantial goal if there are a lot of items to actually obtain, as if there were only 4 items, players would achieve this goal very quickly.

The random generation could also become too boring because a lot of the variation it creates will eventually be repeated. This could potentially be solved by having different levels you can play through, each using a different type of random generation. This could be a potential goal if a lack of variation turns out to be an issue.

1.6 Software Requirements

Overall, the project will use 3 programming languages: C#, PhP and SQL, and will make use of two pieces of software: XAMPP and Unity3D.

During development I will also use two more pieces of software: Blender for 3D models, and VS Code for script/text editing.

1.6.1 Game Requirements

For graphics, I have chosen to use the Unity 3D game engine as it handles rendering 3D scenes and objects internally. This allows me to focus on the main parts of the project: the random level generation and the item database. It can also act as a compiler once the project is finished to create an executable that can be used to play the game and to decrease the overall size of the finished project.

In Unity, objects are handled in a parent-child based “GameObject” system. These “GameObjects” are nodes that can have scripts attached to them such as colliders, 3D models, and custom scripts that are written in C#. I will use this functionality to create the random generation, and to create functioning items for the player to use by attaching the corresponding C# scripts to objects in Unity.

The player will also require a mouse and keyboard as this will be the only supported input method, and also a windows PC to run the game on. They will also need an active internet connection to access their inventories or modify their setups.

1.6.2 Database Requirements

To create functions for the database such as getting user items, verifying logins, and modifying inventories, I have chosen to use PHP. This is because PHP is an extremely versatile language and can be run in most web browsers or as modules in applications.

For the database I have chosen to use MySQL as it is a free and open-source solution for creating and interfacing with SQL databases and tables. To obtain and modify data from this will require SQL queries, which can be created and run through the PHP language.

To host the MySQL database and PHP functions, I will use a program called XAMPP. This allows you to easily set up a localhost server that runs on apache and can run PHP files and therefore have an SQL database.

Finally, to link the database to the game, some C# scripts in-game will make use of the WebRequest function from the UnityEngine.Networking module for C#. This allows the server to be accessed through Unity, and through PHP scripts, the SQL database.

1.7 Hardware Requirements

- OS: Windows 10+ (Unity build for universal windows platform (UWP))
- A computer able to quickly process 3D graphics, albeit simple ones.
- A computer to run the server, or locally run it.

1.8 Success criteria checklist

1. Random Generation:
 - 1.1. Levels are generated with a different layout each time.
 - 1.2. Tiles join neatly with others in their set.
 - 1.3. Every tile has at least 1 path to another tile.
 - 1.4. Level Size is adjustable.
 - 1.5. Rarity of special tiles is adjustable.
2. Database:
 - 2.1. Players can create accounts.
 - 2.2. Players can login.
 - 2.3. Item data can be retrieved.
 - 2.4. Player stats are publicly visible but only changeable by the player.
3. Player:
 - 3.1. Players can move.
 - 3.2. Players can use items.
 - 3.3. Item stats affect how items work.
 - 3.4. Players can retrieve their items from the database.
4. Enemies:
 - 4.1. Damageable stationary targets/enemies.
 - 4.2. Targets move.

- 4.3. Targets track towards the player.
- 4.4. Targets use attacks/abilities.
- 4.5. Anything can be damageable by adding a component.

2.0 Sprint 1 – Player Movement

2.1 Design

This sprint is focused on making the players physical character able to move using inputs. This allows the player to explore the levels and interact with the game. Additionally, this movement system should include semi-realistic physics to allow controlling the character to seem natural. The player should also be able to look around.

The more complex parts of physics will be handled by Unity's inbuilt physics engine. These include friction, air resistance, collision, and momentum. Everything else such as gravity and movement of players or objects will be handled using scripts.

2.1.1 Algorithms

Movement

```

Frame starts
move:
    Take player input w/s and a/d
    Store as Vector3 where w/s is x, 1 to -1, and a/d is z, -1 to 1
    Multiply Vector3 by speed variable.
    Multiply Vector3 by time since last frame.
    Translate player position by Vector3.
jump:
    If space bar pressed:
        apply upward force.
gravity:
    If not on the ground:
        apply downward force equal to gravity.
Frame ends
  
```

Firstly, this algorithm should occur once every single frame of the game to make the movement feel responsive to the players actions as if there is delay it will feel unnatural and sluggish. The input is taken from w, a, s, and d as this is the standard in many games so most players will already be used to it and won't have to waste time learning how to do something as simple as moving.

Secondly, the players input is multiplied by the players' maximum speed. This is because the values taken from the inputs will range from -1 to 1, including decimals. For example, if the backward input is 70% pressed it will read -0.7, meaning 70% of the maximum speed, backwards.

If it were run at this point, and the speed variable was set to 5m/s, it would move 5 metres every frame. To counteract this, you multiply the movement by the time in seconds since the last frame. This accounts for the framerate of the screen and makes

the player move at a constant speed, and at 5m/s and not 5m/frame. For example, if there was 0.1 seconds since the last frame, it would move $0.1 * 5$ metres in that frame.

Additionally, the player should also be able to jump. The space bar is used because it is also a standard so allows players to pick up the controls easily. If this key is pressed, an upward force should be applied, causing the player to jump.

Finally, if the player jumped at this point, they would just continually move upwards from the force. Gravity is used to go against this by slowing the player when they are in the air by applying a constant downwards force. This also provides the effect of falling when nothing is under the player. This force should be added using unity's `AddForce()` method of the [Rigidbody](#) class.

Mouse Look

As the project is in first person, having only movement wouldn't be enough for it to feel natural. Therefore, the player needs to be able to look around with the mouse as well.

```

Frame starts
mouselook:
    Save mouse coordinates
    Take last frame's mouse coordinates
    Calculate the difference between them
    Rotate the player body in z axis by this, and the player head in the x/y axis.
Frame ends

```

Similarly to the movement, this should happen every frame, so it is responsive as possible. Again, it follows conventions for standard controls and uses the mouse position to control it.

The script should find how far the mouse has moved this frame in the x and y axis on the desk, then rotate the player accordingly; mouse x to player body z, and mouse y to player head x/y. The entire player is rotated as you turn left and right as this is natural, however only the camera/head rotates when looking up and down as it would look very weird for the entire player to rotate to look upwards.

Finally, the player rotation could be multiplied by a constant if the rotation is too slow, which could also be multiplied by a percentage "sensitivity" to make the controls customisable for the player.

2.1.2 Key Variables/Structures

Name	Type	Why
Speed	Float	Player speed in metres per second. Float because it will need to be applied each frame and division will be used to find how far to move each frame, probably resulting in a float.
Gravity	Double	Defines the force of gravity to be applied every frame. Double because gravity is 9.81N

Input	Vector3	Defines direction of movement input by the player and will be used to move the player after operations are applied.
Delta time	Float	Time in seconds since last frame. Float as it will be very small.
Jump force	Float	The force to be applied to the player as they jump. Float to interact more easily with other floats.
Vector3	Float	A collection of three floats in the unity engine, x, y, and z. Commonly used to represent 3D coordinates. Vector2 also exists which is the same but with only x and y.

2.1.3 Validation

If any key other than movement keys or space are pressed, nothing should happen (this applies when only player movement scripts are active as other scripts could use other buttons).

If movement keys are pressed, they should do what is expected and only what is expected; w/a/s/d should move, space should jump.

The mouse should be prevented from leaving the game's window on the pc so that the player doesn't accidentally click off the game or click unwanted things on their desktop.

2.1.4 Data to be tested

Test	Expected Outcome
w/a/s/d pressed	Player moves in direction pressed, including diagonals.
Space pressed	Player moves upwards relative to jump force variable.
Gravity	After jumping, player should fall and stop on the ground.
Speed var	If it changes, the player's actual speed should change to match.
Jump force var	If it is increased the player should jump higher, vice versa.
Mouse left/right	Entire player should rotate in the same direction as the mouse.
Mouse up/down	Only the head should rotate in the same direction as the mouse.
Sensitivity	Increasing/decreasing should increase/decrease rotation speed.

2.2 Development

2.2.1 Movement Script

Ground Check

This function is created first, as it is used to identify whether or not the player is able to modify their movement at that point in time, and what type of jump they should use, if they have one. It does this by identifying if the player is touching the ground.

```
private void GroundCheck()
{
    if (Physics.CheckSphere(transform.position - new Vector3(0, groundCheckDistance, 0), 0.1f, groundLayer)) { isGrounded = true; }
    else { isGrounded = false; }
} //This checks if the player is touching the ground. Sets public variable isGrounded so other functions/scripts can see this.
```

Firstly, this uses unity's Physics class to project a virtual sphere from a point at the bottom of the player's body. This sphere then identifies any objects that are inside it that are part of the "ground" layer. This layer is defined in unity and applied to every object that should be considered as the ground. If this sphere function returns any objects at

all, it sets the “isGrounded” variable to true. This variable is public meaning that other scripts will be able to access it and also other functions from this, the player movement script.

Jumping

This uses one main function to make the player jump, along with smaller sections to identify when the jump is valid and what kind of jump to use (double when in air, normal when on ground).

```
private void Jump()
{
    if (jumpCharges > 0)
    {
        //Reset vertical velocity
        if (playerRB.velocity.y < 0) { playerRB.velocity = new Vector3(playerRB.velocity.x, 0, playerRB.velocity.z); }

        if (!isGrounded)
        {
            //Jump (in air)
            playerRB.AddRelativeForce(0, jumpHeight, 0);
            isJumping = true;
            jumpCharges--;
        }
        else
        {
            //Jump (on ground)
            playerRB.AddRelativeForce(0, jumpHeight / 1.5f, 0);
            isJumping = true;
        }
    }
}
```

This is the main function that is called whenever a valid jump input is made. Firstly, it resets the vertical velocity of the player. This is because without this, the player would be able to press the jump button multiple times quickly and expend all of their jumps which would combine in velocity to create an excessively high single jump. Resetting this means jumps actually have to be spaced out with good timing to get the most height and are also more predictable. Furthermore, this means double jumps still function when falling extremely quickly as without this they would only slow your fall slightly and not actually make you jump upwards.

Next, there is a check on the “isGrounded” variable to see which type of jump should be used. If the player is on the ground, a normal jump is used which is slightly weaker than an in-air double jump due to the character having a small set of jets to jump, but only when in the air. If the player is not on the ground however, it uses another jump that has slightly more force due to the characters “jump jets”. However, this jump is limited as infinite would allow the player to simply fly. It uses a variable “jumpCharges” to ensure the player can only jump a maximum number of times. When this is zero, the player can’t jump, and when the player jumps in the air it is incremented by -1. It is also reset whenever the player touches the ground to its default value which varies based on the player’s “agility” statistic.

```
//Status updates
if (isGrounded) { jumpCharges = jumpChargesBase; }
```

Resetting the players jump charges is seen here, which occurs every frame along with a call to the ground check function.

```
private void ResetDefaults()
{
    //SetValues Based On Agility
    jumpChargesBase = 1 + (player.agility / 50);
    movementSpeed = baseSpeed + ((baseSpeed / 2) * (player.agility / 100));
    jumpHeight = baseJump;
}
```

Finally, this function is called every 20 frames, and it changes the players default values such as its movement speed, jump charges, and jump height. It refreshes this often in case the player's stats change due to gear being changed or levelling up (this will be implemented later). It doesn't refresh every frame though as this would be unnecessarily often and calling it every 20 frames is 20 times more efficient while having almost no noticeable effect on when the values actually update in real time.

Moving

This is the main movement script and is called frequently to provide the maximum responsiveness to the controls. However, it isn't called every frame, but instead in unity's fixed update function. This means that it is called at a far more consistent rate, which reduces any physics bugs caused by inconsistently changing the player's velocity.

```
private void Move(Vector3 movement)
{
    movement = movement.normalized; //Stop the velocities combining when doing a diagonal (two key) input.
    movement = playerRB.rotation * movement; //Change movement to be relative to facing direction.
    movement = movement * movementSpeed * Time.fixedDeltaTime;

    if (isGrounded || isJumping)
    {
        previousMovement = movement; //Set this frames inputs as the last valid input.
        playerRB.MovePosition(playerRB.position + movement);
        isJumping = false;
    } //Use this frames movement inputs to move if the player is on the ground.
    else { playerRB.MovePosition(playerRB.position + previousMovement); }
    //Use the last valid frame's movement inputs to move if the player isn't grounded.
}
```

Firstly, it takes the movement input, which is a 3D vector where the values x and z vary from 1 to -1 based on the input key that is pressed. This vector is then normalised which means that if a diagonal input where to happen it would still move at the same speed as if it was a single directional input. This wouldn't happen without it however, as a diagonal input requires two keys, and the overall input magnitude would be 2, as two

keys being pressed both add one to the vector. Normalisation divides the input strengths by the amounts of inputs pressed; if w and d are pressed, instead of being 1 + 1 with a magnitude of 2, it is 0.5 + 0.5 with a magnitude of 1 which is the same as if only w was pressed, except it is in a diagonal direction. This would work with a 2D vector but translations in unity take a 3D vector as an input, so this avoids converting it.

Next it multiplies this by the players facing direction. This is a 3D quaternion, which when multiplied with a 3D vector, makes the vector relative to that rotation. Overall, this means when you press w you move in the direction the character is facing instead of north.

Next it checks if the player is either on the ground or currently jumping. If this is true it applies the movement to the player by using unity's move position function to translate the player by adding the movement vector to the player's current position.

If the player is not on the ground or jumping, it adds the last valid movement to the players position, which creates the effect of the player seeming to maintain velocity when it is in the air.

2.2.2 Mouselook Script

Getting references

First, the game object that is the player's head and the object that is its body must be obtained so that they can be rotated properly.

```
private void Start()
{
    //Get object references.
    Body = transform;
    Head = Body.GetChild(0).GetChild(0) as Transform;
}
```

This is done by searching the hierarchy instead of taking them by a hardcoded reference as it allows them to be dynamic as long as the object for the head is kept as the second child of the body. Therefore, if the player's model were changed at any point from just a white cylinder, the code wouldn't have to be modified.

Firstly, the body is taken and assigned. This is simply taken to be the transform of whatever object this script is a child of. This object will be rotated with the mouse's horizontal movement as it gives the impression of the whole player turning round to look.

Next, the head is assigned. This will be the second child of the body object, and the camera itself will be attached to it. This means that whenever the body rotates the head and camera rotate with it, and whenever the mouse is moved vertically, only the head and camera rotate vertically so the entire player doesn't start levitating weirdly.

Taking inputs

Every single frame, the position of the mouse is recorded and stored as the input. Its x and y positions are stored separately so they can be applied to the player's rotation separately.

These inputs are also each multiplied by the time since the last frame so that the controls feel consistent across inconsistent framerates. They are also multiplied by the sensitivity value which is public so it can be modified later to allow players to customise their controls.

Rotating the player

```
mouseY = Mathf.Clamp(mouseY, -90, 90);
//Sets max rotations so the player cant look past fully up or fully down.

Head.transform.localRotation = Quaternion.Euler(mouseY, 0, 0);
//Rotate only the head in the x axis (around the horizontal).
Body.transform.localRotation = Quaternion.Euler(0, mouseX, 0);
//Rotate the whole player
```

The player is rotated immediately after inputs are taken, in the same frame to minimize any input delay.

Firstly, the value for the mouse's vertical position is "clamped" which means that it cannot exceed 90 or drop below -90. What this actually does is mean that the player cannot keep rotating their view upwards or downwards once they are looking completely up or down respectively.

Finally, the rotations are applied directly to the corresponding objects, horizontal to the body and vertical to the head. This is done directly without any further checks as nothing should affect the player's ability to look around as this would be disorienting and most likely not be fun at all for the player.

2.3 Testing

2.3.1 Tests

ID	Description	Inputs	Expected Outcome	Actual Outcome	Changes
1	Moving	w;a;s;d	move forward; move left; right; back	Forward; right; left; back	Yes: Invert a & d.
2	Jumping	Space	Player should move upwards with high initial velocity, then slow and fall.	Player moves upwards then falls but goes through the floor.	Yes: Make the floor thicker.
3	Gravity	Jump/Walk off a ledge	Player should fall at a rate that looks like reality (about 10ms^{-2})	Player falls at an increasing rate that looks natural and is consistent each time.	No
4	Changing speed	Set speed to 1000;	The player should move extremely fast; The	Player goes so fast they seem to teleport; The player moves at a	No

		set speed to 3.	player should move at a normal speed (3m/s).	moderate walking style speed.	
5	Looking around (Horizontal)	Move mouse left and right.	The player should rotate its entire model in the same direction as the mouse, and by a directly proportional amount.	Only the camera rotates, not the body, however it rotates in the correct direction and by the correct amount.	Yes: fix references
6	Looking around (Vertical)	Move mouse up and down.	Only the player head/camera should rotate, but in the same direction as the mouse and by a directly proportional amount.	The camera and head rotate together and proportional to the mouse movement. However, it is inverted to the direction of the mouse movement.	Yes: invert rotations.
7	Sensitivity	Set to 5000; set to 50	The player should look around extremely fast with only small mouse movements; the player should look around at a speed much closer to how far the mouse moved.	The camera moves so fast it is almost uncontrollable; The camera still moves fast but a lot slower.	Small: set to 30 instead of 50.

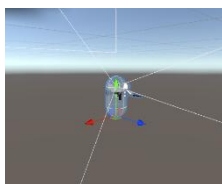
2.3.2 Fixes

[1] Inverting sideways movement

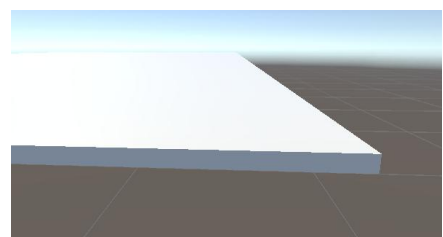
```
moveInput = new -Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"))
```

When the input is taken, the horizontal input simply has a negative added in front of it. This means if it is negative already it becomes positive and vice versa, hence inverting the direction of the horizontal movement.

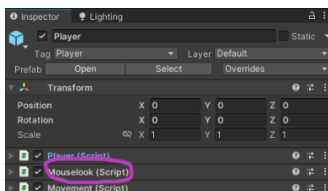
[2] Increasing floor thickness



Previously the floor was a plane which has no thickness or collision, so the player fell through. To combat this, it was replaced with a cube that was resized to be the same size and shape except with a thickness of 1m to prevent anything falling through it.



[5] Fixing player rotation references



The Mouselook script was a child of the camera object as originally, I thought it should go here due to it being to do with moving the camera. Instead, it was moved to be a child of the player's root.

[6] Inverting mouse look

```
Body.transform.localRotation = Quaternion.Euler(0, -mouseX, 0);
```

A negative sign is simply added before the rotation variable when the rotation is applied.

2.4 Sprint Overview & Maintenance

In this sprint the goal was to make the character controllable by the player, enabling them to interact with the world by moving and looking around, and in a way that was responsive.

The ability for the player to move using W,A,S and D keys was created using a script that takes the key inputs and moves the player based on their orientation, stats, and what surface they are on (air or ground).

Looking around was created by taking the movement of the mouse in that frame and rotating the player's head and body accordingly, relative to the mouse movement and affected by a customisable sensitivity value.

There were no major issues, and only small things had to be fixed such as inverting directions and ensuring that scripts were affecting the correct objects.

Next should be creating random level generation functionality so that the player has something to move around.

3.0 Sprint 2 – Level Generation

3.1 Design

The levels should be generated randomly, but in a way that makes them distinctly part of a set that would not look out of place next to each other. To achieve this a simpler random generation system based on arranging premade tiles in random configurations will be used. This will create familiar levels that are easily changed with level designs that can also be replayable and ideally not boring due to them being slightly different each time.

These levels should all be generated using a single script that can work with multiple different sets of tiles to create different styled levels using a single method. However, more random generation scripts could be added in the future to work with those same tile sets or even something completely different if a drastically different level were needed.

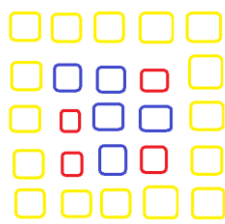
3.1.1 Algorithms

There should be 3 different types of tiles that can be generated, each using different rules of where to generate. Normal tiles that can go anywhere except edges, edges that only go around all other tiles, and rule tiles that can only be generated next to specific tiles.

Identifying Tile Types

There should be three tile types:

- Edge: Can only generate at the edges of the level, e.g: walls/cliffs/mountains.
- Rule: Can only generate next to specific tiles, e.g: structures/lakes/chests.
- Middle: Anywhere. Should look natural next to any other middle tile in the set.



They should also generate in this pattern where yellows are edges, reds are rules and blues are middle.

The tiles for each level “set” will be stored together in one folder so their type should be easily identifiable by the script. To do this a good method would be by adding a prefix to their name to identify this. This method would be efficient due to the script only having to check a single string; their name, rather than identifying the tile from other characteristics.

One issue with this method could be that if the prefixes are forgotten then the tiles will not generate at all or could generate in the completely wrong locations. However, it would be obvious if this were missed during development, so it is not a major issue.

Generating middle tiles

```
Create a grid/array
Set grid to desired level size (in tiles)
Subtract one from each direction of the grid size
Set "position" to 0,0 (top left).
Loop While xPosition < grid width:
    Loop While yPosition < grid height:
        Pick random tile from list of middle tiles
        Create an instance in unity
        increase yPosition (go downwards)
    increase xPosition (go across)
    set yPosition to 0 (go back to top of column)
```

Firstly, the size of the grid and level type should be specified as an input because this allows the generator to know how big to make the grid and which set of tiles to use.

Next, the script should create a virtual grid as an array that it will iterate through. It should be the size of the specified level size. E.g: if the level wants to be 5 tiles across, the array should be a 5*5 array, 5 wide, 5 high. However, it should be 2 less width ways and height ways to make room for edge tiles.

Finally, the script should iterate through this grid, choosing a random tile from the list of middle tiles from this set and creating an instance of its prefab in the corresponding world position in unity. For example, if it is at position 1,5 on the grid and each tile in the set is 5m², the tile should be created in unity at position 5,0,25.

Generating edge tiles

These will follow an algorithm almost exactly like the middle tiles. It will follow the same procedure except for two differences. When the middle tiles iterate through the grid, the

edge tiles grid will be 2 tiles larger in width and will iterate only along the edges of the grid. It does this by only resetting the xPosition and yPosition of the virtual grid every other loop in an alternating pattern as this will cause it to go around the edge.

Generating rule tiles

```

Loop
    i = 10
    while i > 0:
        Pick a random tile from the list of rule tiles.
        Check prefix and find out its index in Rules.txt
        Generate 2 random numbers between 0 and gridSize-1 as coordinates.
        coordinates by tile width.
        Raycast a sphere around this point to return the tile in this location.
        If its prefix = the current rule tile's index:
            Replace with rule tile.
            i = 0
        else
            i -= 1
    Repeat (grid size / 5).

```

Firstly, the entire process of generating tiles will be done a number of times proportional to the grid size. This means that if a levels size is bigger, it will have more special rule tiles, and smaller grids won't be overwhelmed by them.

Secondly, when a rule tile is randomly chosen to be generated, it is given 10 attempts to generate. This means that the amount of rule tiles won't always be the same, but it is likely that there will always be one or two. If a rule tile succeeds in generating, it moves on to the next one (if there is a next one).

To actually generate the tiles, the randomly chosen tile's prefix will be located in the Rules.txt file for that tile set. The location of it will be saved. For example, if the prefix was “,” and was the tenth one along in the text file, its index would be 10. This value is then compared to the prefix of a randomly chosen tile that has already been generated. If they are equal, then the tile is replaced. If they are not equal, then nothing happens, and that rule tile is given another attempt at generating (if it has any remaining).

3.1.2 Key Variables/Structures

Name	Type	Why
X/Y position	integer	Tracks the position currently assigning a tile.
Middle Tiles	List<Object>	List of all middle tiles in the current set.
Edge Tiles	List<Object>	List of all edge tiles in the current set.
Rule Tiles	List<Object>	List of all rule tiles in the current set.
Grid Size	integer	The desired size of the level to be generated.
Level Set	string	Dictates what level set the generator uses. Match name to directory in './resources/{name}'.

Prefixes

All prefixes should be come at the start of the name and be followed by a ‘,’ character so they can be very easily identified using the Split({string},{character}) function in C#.

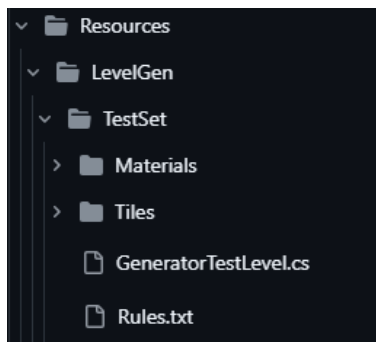
- Edge tiles: ‘e,{name}’
- Rule tiles: ‘{f-->o},{name}’
- Normal tiles: ‘{number},{name}’

3.1.3 Data to be tested

Test	Expected Outcome
Tile set name var	Changing the tile set name in the script should change which folder in resources all tile prefabs are taken from.
Tile locations	Edge tiles should be around the edge, middle tiles should be everywhere else, rule tiles should be scattered throughout and next to the correct middle tiles. No rule tiles should be on the edge.
Grid Size	Increasing this or decreasing this should change the size of the generated level accordingly.

3.2 Development

3.2.1 Resources



All random generation scripts will be generating by choosing randomly from given asset groups, so they need to be accessible by the code. To do this, all the tiles can be stored in a folder called “resources” inside of project/assets. This corresponds to a function in the unity engine that looks for a “resources” folder and includes all these files in the final build uncompressed so that they can be accessed by scripts.

The tiles themselves will be square prefabs that will all be the same size in a set. However, they are split into folders based on which set they belong to. Additionally, the tiles will have rules about when they can generate so to identify which rule they use they will have a prefix separated by a “,” that tells the script which rule to use when placing it.

3.2.2 Level Generator

The point of this script is to generate a level using a set of rules. It does this by generating tiles, choosing random ones from each defined group, a group for edge tiles and a group for central ones. This allows levels to seem distinctly different by assigning them a different set of tiles with a different theme, and the same levels will be recognisable from using the same tile set but different each time due to the random arrangement.

Loading Tiles

```
private void LoadTiles()
{
    //Takes tile prefabs from stated folder and adds them to an array.
    tiles = Resources.LoadAll("LevelGen/" + GeneratorTileset + "/Tiles");

    List<Object> edgeTilesList = new List<Object>();
    List<Object> normalTilesList = new List<Object>();
    //Separate array for each group of tiles (edge and center).

    for (int i = 0; i < tiles.Length; i++)
    {
        switch (tiles[i].name.Split(',')[0])
        {
            case "e": edgeTilesList.Add(tiles[i]); break;
            default: normalTilesList.Add(tiles[i]); break;
        } //Identifies which group a tile should go in based on name prefix.
    }

    edgeTiles = edgeTilesList.ToArray();
    normalTiles = normalTilesList.ToArray();
}
```

This function runs whenever an object with this script attached is activated in the scene. Its purpose is to find out which tiles are to be used for this specific level before generating it. As all the tiles are stored in the same place, “./assets/resources/levelGen/{tile_set_name}”, it only has to take the tile set name as an input. It will then open this folder and add all prefabs inside it to the lists in the script. The tiles aren’t just added randomly however, as they all have prefixes to their name separated by “,”. If the prefix is an “e” they are added to the edge tiles group and if there is no prefix, they are assigned the default group. This also allows for further expansion of the code by easily adding more groups if more rules are added, which can be done by adding an additional list, and another case for the prefix character before the default in the switch statement.

Middle Tiles

This is part of the generate() function that can be called at any point externally as it is a public function allowing it to be controlled by things such as checkpoints or sections that load into new levels. This part of the function generates tiles based on the rule that they aren’t on the edge.

```
#region Generate Middle Pieces
GridSize -= 1;
float CellSize = SpawnTile.transform.localScale.x * 10;
float currentX = 0 - ((GridSize + 1) * CellSize) + CellSize;
float currentY = 0 + ((GridSize + 1) * CellSize) - CellSize; //Starts generation in top left corner of map
```

Firstly, it takes the size of each tile by reading the prefab data, and also the desired grid size (in tiles) as inputs. Next it sets two variables that store the location to generate the next tile as the top left of a virtual grid. Each square in this imaginary grid is the size of one single tile (or cell), so the script sets these to the top left of this grid where the centre of the grid is at 0,0,0 in unity. However, this “grid” is one smaller than the inputted grid size as these are only central tiles and should leave space for the edge tiles after.

```
while (rows >= 0)
{
    while (cols >= 0)
    {
        if (currentX != 0 || currentY != 0) { Instantiate(normalTiles[rnd.Next(0,normalTiles.Length)], new Vector3(currentX, 0, currentY), Quaternion.identity); }
        currentX += CellSize;
        cols--;
    }
    cols = (GridSize * 2);
    currentX = 0 - ((GridSize + 1) * CellSize) + CellSize;
    currentY -= CellSize;
    rows--;
}
#endregion
```

Next, it goes through the grid one by one following these steps: first it chooses a random number using C#'s random function within range of the list of valid tiles for the middle and chooses that tile. It then creates an instance of the corresponding prefab in unity at the current coordinates of currentX and currentY. Then it moves along the grid by an amount equal to the width of 1 tile and repeats. It does this until all tiles are filled in the grid.

Edge Tiles

These tiles are generated according to the rule that they must be on the edge.

Again, a virtual grid is created in the same way and started in the top left, however this time the grid size is exactly the specified size, not one less as it includes edge tiles.

```
for (int i = (GridSize * 2) + 1; i >=0; i--)
{
    Instantiate(edgeTiles[rnd.Next(0, edgeTiles.Length)], new Vector3(currentX, 0, currentY), Quaternion.identity);
    currentX += CellSize;
}
```

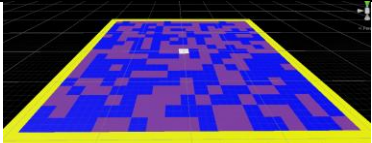
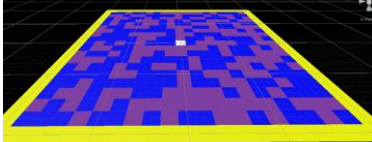
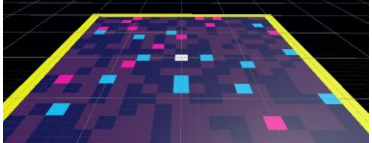
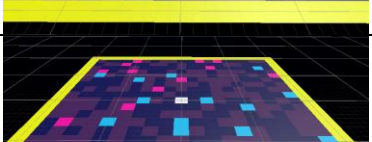
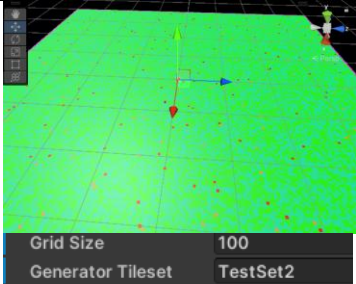
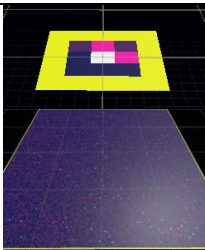
Next there are 4 for loops that each go along the sides of the grid generating a random edge tile. They do this by iterating through the current X and Y coordinates on the virtual grid. For example, in the above code it increases the X each loop, meaning that it moves to the right 1 tile each time. In the next loop it would decrease the Y, hence moving downwards. This would repeat accordingly until the edges are filled.

[WIP] Rule Tiles

3.3 Testing

3.3.1 Tests

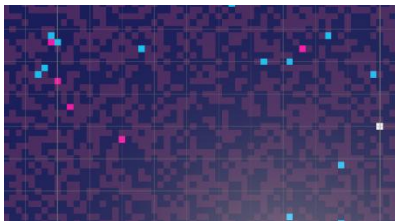
For all of the tests, a prototype tile set is used with flat colours, so it is easy to tell which tiles have generated and if they generated correctly.

ID	Description	Inputs	Expected Outcome	Actual Outcome	Changes?
1	Middle Tiles	Run script	Middle tiles (blue and purple in test set) generate in a square of width 1 less than grid size.		No
2	Edge Tiles	Run script	Edge tiles (yellow in test set) generate around the edge of middle tiles).		No
3	Rule Tiles A	Run script	Rule tiles (pink in test set) might generate a couple when next to at least 2 purple tiles.		Yes – Compare prefix to surrounding tiles, not the actual tile.
4	Rule Tiles B	Run script	Rule tiles (light blue in test set) might generate a couple when next to at least 2 blue tiles.		Yes – Compare prefix to surrounding tiles, not the actual tile.
5	Tile set	Change tile set name variable	Script should look for a folder in resources/generation/ with that name and generate using that set instead. If there's no folder with the name, it should fail to generate anything, and the player fall through the floor.		No
6	Grid size	<div>Grid Size 2</div> <div>Grid Size 100</div>	Increasing should increase the size of the generated level. Decreasing should decrease the size of the level. Unit of measurement is tiles.		No

3.3.2 Fixes

[3&4] Change comparison

Changed the raycast to check the surrounding 4 tiles in each cardinal direction instead of the tile it was replacing. This means it checks what its next to instead of what is about to be irrelevant as it will be gone.



This slightly lowers the amount of rule tiles but this is good as there were previously far too many even with a low multiplier.

```
RaycastHit hitTop;
Physics.Raycast(chosenPosition + new Vector3(0, 0, CellSize), -Vector3.up, out hitTop);
RaycastHit hitRight;
Physics.Raycast(chosenPosition + new Vector3(CellSize, 0, 0), -Vector3.up, out hitRight);
RaycastHit hitBottom;
Physics.Raycast(chosenPosition + new Vector3(0, 0, -CellSize), -Vector3.up, out hitBottom);
RaycastHit hitLeft;
Physics.Raycast(chosenPosition + new Vector3(-CellSize, 0, 0), -Vector3.up, out hitLeft);
//Find the tile in the chosen position

int adjacentMatches = 0;
if (ruleIndex.ToString() == hitTop.transform.name.Split(',')[0])
{
    adjacentMatches += 1;
}
if (ruleIndex.ToString() == hitRight.transform.name.Split(',')[0])
{
    adjacentMatches += 1;
}
if (ruleIndex.ToString() == hitBottom.transform.name.Split(',')[0])
{
    adjacentMatches += 1;
}
if (ruleIndex.ToString() == hitLeft.transform.name.Split(',')[0])
{
    adjacentMatches += 1;
}

if (adjacentMatches >= 2)
{
    Debug.Log("Matches, creating rule tile.");
    RaycastHit hit;
    Physics.Raycast(chosenPosition, -Vector3.up, out hit);
    Destroy(hit.collider.gameObject);
    Instantiate(chosenTile, chosenPosition - (Vector3.up * 10), Quaternion.identity);
    Debug.Log("Rule tile created.");
    //Swap out current tile if it fits the rule
}
```

3.4 Sprint Overview & Maintenance

In this sprint the goal was to create a universal script that could be used to generate all variations of levels needed. It should use the same algorithm for each, and be able to modularly swap out the sections it generates with to create different, curated but random levels.

The random generation script was developed around one single set of tiles, where it places them according to a set of rules. To enable this to work with multiple levels, the rules that were originally hardcoded were changed to a text file, allowing them to be modified based on the level style. Also, instead of having direct references to specific level elements, it creates its own references whenever an instance of the script is made, using a single parameter to identify what folder to look in.

In conclusion the designed system was implemented as planned, and works effectively, needing no changes. There were some small bugs in the final code, but these were only minor so could be fixed quickly.

4.0 Sprint 3 – SQL Server

This sprint is focused on creating a normalised database to store all the items the player can obtain together. It should also create the ability for this database to be interacted with by external scripts such as the game itself. Finally, this sprint should also aim to store player data and associate items with that data to form an inventory system for each player.

4.1 Design

This sprint is focused on creating the database that allows the players to continue their version of the game from anywhere, and to be competitive about their progress.

4.1.1 Algorithms

Login

The player will be able to log in to a created account using a php script hosted on the same location as the SQL server. This php script should take the player's inputted username and select the row with that username in the SQL table "users". It should then take the inputted password and compare it to the password stored in that same row. If they match it should return a successful login attempt, if they don't, then it should fail.

This function should be run whenever a player wants to take an action that would affect their own account such as changing their inventory, their equipped items or obtaining new items. To avoid the player having to enter their information every time, they will be asked to log in once at the start of the game and if it is a successful attempt, the login information will be saved to the player script privately. This is secure as the player script in unity is the script that will use the login function so the data can be kept encapsulated to that single script.

```
input username & password
select the password from the sql row where the username is equal to the input.
if sqlPassword = inputPassword:
    output "successful login"
else:
    output "failed login"
```

To call this php script in unity, it will use the UnityWebRequest function that allows data to be retrieved from websites. This will be used to call the php script and input the stored login details in the player script into the php script. The output of the php script will then be returned as the output of the UnityWebRequest function.

Login -> Store in player C# script -> UnityWebRequest -> php -> Player C# script

Create User

This will be used to create an entry for a new account in the 'users' table. This will be done in the same way as the login script: using a php file hosted in the same place as the database.

```
input username & password & repeat of password
check each field has a value of a minimum length (eg: 3)
if not:
    end script and output: "enter values"
check that the two password fields match
if not:
    end script and output: "check passwords match"
select any users from the sql table where the name matches the inputted username
if this returns more than 1:
    end script and output: "username taken"
if script still running:
    insert new entry into sql table with inputted username and password
    if database has the new entry:
        end script and output: "success"
    else:
        end script and output: "fail, try again"
```

First it should take the username the player wants, and also the password they want. However, the password should be repeated to ensure they didn't make a typo. The script should then check that none of these inputs are empty and are of a minimum length so that the password isn't too short. It should then ensure that the password inputs match each other.

If both of these checks pass, it should then request any user entries from the database where the name is the same as the inputted username. If any are returned the script should fail and tell the user that the username is already taken.

Finally, if every check is passed, a new entry should be made in the 'users' table with the entered username and password. This entry should then be requested from the database to check it was added successfully. If it was the script should end saying "success", or with "fail, try again for a failure.

Getting Player Data

The player information such as their inventory and equipped items should also be able to be obtained at any time. This will be obtained from the SQL database using another php script that retrieves specified information from the player id that is currently stored.

It should also call the login function at the start of itself and use the stored login details as this prevents players from modifying other players data.


```

call Login.php
if successful:
    SELECT * from users for the current user
if unsuccessful:
    Only select publicly available stats like username and xp
output all data segments, separated by ','

```

The reason it still gets data if the password is wrong, is because some data should be accessible publicly to allow friends to see each other's stats and be competitive with each other.

This script is also a php scrip and will be called in the same way as the Login php script: using UnityWebRequest.

Getting Player Inventories

This script shouldn't have a password check as this only reads data about the player, nothing is modified. Additionally, the data should be able to be seen by other players so that they can be competitive about the items they have.

It should work by taking a player's ID and looking in a table that contains player IDs next to item IDs. This would be a table that connects the table of items and users together creating a relational database to avoid many entries of the same data.

It should then return the ID of any items that are owned by the specified player (item IDs on the same rows as their player ID). This data should then be reformatted into a string, so it is as compatible as possible meaning it saves time when creating things that use the function.

Finally, the data returned should only be data that is unique to the player's version of the item and its ID, as any non-unique data can be retrieved using the item's ID from the table of items. Unique data includes things such as the level of the one the player found as this makes theirs better or worse than someone else's version of the same item. This could also include things like applied cosmetics if they ever ended up being added as it leaves room for scalability such as that.

Getting Item Data

This script should retrieve the data about an item that is always constant for that item, i.e: the data in the table of items. To do this it should simply take an item's ID as an input and then try and find it. If it finds it in the table, it should return the data as a string for maximum compatibility, and if it fails it should output an error message.

4.1.2 Prototyping / Normalisation

All the data

Player Name	Password	Date Created	Items Inventory	Level	Speed	Strength	Item Stats
Random guy	guy123	221124	stick, sword, gun	0	1	1	sword;range1: damage5,gun;range10: damage1

What should be stored is the player's name and password for logins. It should also store the player's stats. These include level, speed, and strength that affect the player's gameplay; level scales the level of items they find, speed increases the move speed and number of jumps, while strength is a multiplier applied to using swords. Finally, it should also store the items the player has along with their stats.

1NF

PlayerID	ItemID	name	type	speed	strength	range	rarity	level
0	0	stick	object	0	0	0	1	1
0	1	katana	sword	2	10	1	3	1
0	2	pistol	gun	100	5	50	5	1
ID	Player Name	Password	Date Created	Level	Speed	Strength		
0	Random guy	guy123	221124	0	1	1		

This makes it so that each cell of data only contains one value. To do this, a second table, 'items' is created, and the 'id' column

is added to the original table called 'users'. Each row of the 'items' table contains the name and stats about that item, while the first two columns identify which user owns that item. To retrieve an item, you would request that item ID from the row that starts with the correct player's ID.

2NF

PlayerID	ItemID	Quantity	Level			
0	0	1	1			
0	1	1	1			
0	2	1	2			
1	2	2	1			
ID	Player Name	Password	Date Created	Level	Speed	Strength
0	Random guy	guy123	221124	0	1	1
1	R2nd2m g2y	guy124	221124	0	1	1
ItemID	name	type	speed	strength	range	rarity
0	stick	object	0	0	0	1
1	katana	sword	2	10	1	3
2	pistol	gun	100	5	50	5

Previously, in the items table, for any two rows that had the same item ID, everything would be the same except for the level and player ID. Therefore, the values that change can be put in a separate table, 'useritems' that defines what user has which item. Meanwhile the items table only contains constant data about each item that can be referenced with its ID.

However, the useritems table needs a fourth column, quantity, as this prevents multiple identical entries being created when a user gets two of an item.

4.1.3 Validation

As php scripts interact with the database directly, and it contains other player's data not just the current player's data, any scripts that modify or change data, or access protected data, should only function if a password check is passed.

For example, the script to get data about a player has two possible outcomes. If the password entered at the beginning of the script matches that of the player whose data is being accessed, then all data is returned. If the password doesn't match or just isn't entered, the script only returns data that should be publicly accessible to make the game competitive such as the username and experience level.

Secondly, any data entered into the database must be checked that it is acceptable and in the correct format or it will be entered into the database incorrectly. Examples of this

include the repeat password field when creating a new user as this prevents an incorrect password caused by a typo from being accepted.

4.1.4 Key Variables/Structures

Name	Type	What for
Input Username	String	Used for logins.
Input Password	String	Used for logins.
Player/Item ID	int	Used to get data about that thing from database.

Player Data Class

When the player successfully logs in the login information will be saved to a Player class in unity that allows the login function to be checked when needed without the player having to repeatedly login. This class should also contain player data such as, XP, the player ID, and the login information.

Name	Type	Status	What for
Username	String	Private	Used when login Function is repeated.
Password	String	Private	Used when login Function is repeated.
ID	int	Public	Used to obtain player data such as the inventory.
XP	Int	Public	Used to scale the strength of loot rewards and strength of enemies.

Player/Item stats

All items will use the same 3 stats as this allows all the items to be stored in the same table, simplifying systems and reducing the storage space needed. These three stats will do something different depending on the items 'type'.

This also allows new item types to be created without needing to create entirely new data points, as the existing ones can be used.

Items also have other stats such as rarity and level, but these mean the same across everything.

.	Speed	Strength	Distance
Player	Movement speed	Melee damage multiplier	
Junk			
Melee	Swing/attack speed	Damage per attack	Reach
Hitscan	Fire rate	Damage per bullet	Range
Projectile	Fire rate	Damage per bullet	Projectile speed
Health	Use speed	Health restored	

SQL Database Tables

'users'

id	username	password	date	experience	speed	strength
int	string	string	Int: ddmmyy (date created)	Int - used to calculate player's level.	int – stat,	int – stat

‘items’

id	name	type	speed	strength	distance	rarity
int	string	string	Int – stat	Int – stat	Int – stat	int – how likely item is to drop

‘useritems’

Player_id	Item_id	quantity	level
Int – id of the player that owns this entry.	Int – id of the item being referred to.	Int – amount of the item the player has.	Int – the level of this instance of the item.

4.2 Development

4.2.1 PHP Scripts

Overall

All php scripts start by creating a connection to the SQL server so that data can be modified in it. Depending on the script, it will create a connection with different permissions. It will either have read permissions, write permissions or both.

```
//Create connection
$conn = new mysqli($servername, $username, $password, $dbname);
//Check connection
if($conn->connect_error){
    die("Connection failed: " . $conn->connect_error);
}
```

The connection is firstly created using the server details that correspond to the correct permissions for the script. For example, if it has read and write permissions it will use “root” as the \$username for the server with the corresponding \$password.

It then checks for any errors in the connection. If there are errors, the \$conn->connect_error variable will not be null. Therefore, if it has a value, the die() function can be used to end the script, while outputting the error. For example, if the server is offline, it will stop the script and output something like “connection timed out.”.

If there are no errors, the rest of the script is allowed to continue.

Login

```
//User details
$username = $_POST["loginUsername"];
$password = $_POST["loginPassword"];
```

Firstly, the php script takes the username and password that the user entered using post variables. This means that the variables are assigned using the parameters that the web request was made with. In unity this corresponds to the WWWForm data type, which is sent along with a web request, and it defines what these two variables should be.

```
//SQL
$sql = "SELECT password FROM users WHERE name = '" . $loginUsername . "'";
$result = $conn->query($sql);

//Check if login details match
if($result->num_rows > 0){
    while($row = $result->fetch_assoc()){
        if($row["password"] == $loginPassword){
            die("Login Success!");
        }else{
            die("Check Login.");
        }
    }
}else{
    die("Check Login.");
}
```

Next, an SQL statement is created using the user inputs, and it is used to retrieve the password associated with the username inputted.

It then checks if the SQL query returned anything at all. If it did not, then the user does not exist, so the script ends itself and outputs to check the login. This is treated by unity as a failed login attempt.

If the query does return something, this value is compared to the inputted value for the password. If they match exactly, it ends with “Login success” and if they don’t match it is treated as a failure.

Create Account

Similarly to the Login script, this script takes the user’s desired username and password using the POST method for web requests. However, it also takes a second password input to ensure the user actually typed the right thing.

```
//Validation
if($createPassword != $repeatPassword){
    die("Passwords must match.");
}
if(strlen($createUsername) < 3 || strlen($createPassword) < 3){
    die("Username and password must be at least 3 characters.");
}

//Check if user with username exists
$sqlCheckName = "SELECT name FROM users WHERE name = '" . $createUsername . "'";
$checkNameResult = $conn->query($sqlCheckName);
if($checkNameResult->num_rows > 0){
    die("User with that name already exists.");
}
```

The first thing the script does is validation, to see if the user's desired information is valid. It uses an if statement for each check, where if the check fails (is true) it will stop the script with an error message. Hence, preventing an invalid user being created.

The first check simply compares the two password inputs to see if they match. If they match it fails and tells the user to make sure their passwords match. This is done because it prevents the user doing a typo when creating their password.

The second check makes sure that there actually is a username and password entered, and that they are both past a minimum length. Passwords shouldn't be shorter than 3 characters because they would be guessed far too easily, and usernames cannot be blank, and 2-character usernames look dumb.

The third and final check uses an SQL statement to retrieve any existing users with the desired username. If the value this returns is anything other than null, the username has been taken so the script fails and tells the user to pick a different name.

```
//SQL for creating user
$sqlCreateUser = "INSERT INTO `users` (`name`, `password`, `date`, `xp`, `clanid`) VALUES ('" . $createUsername . "', '" . $createPassword . "', '" . date("dmy") . "', '0', '0')";
$creationResult = $conn->query($sqlCreateUser);

//Outputs
if($creationResult == 1){
    die("Account created.");
}else{
    die("Failed.");
}
```

Finally, if the script is still running at this point, it means the data entered by the user is valid, so an entry for their account can be made. This is done using an SQL statement

that inserts a new entry into the 'users' table. This data inserted includes the desired username and password, the current date, and initialisation of values to be used later.

After this has run, the database should return the number of entries made. If this number is 1, the user was successfully created and the script ends. If it is less than one, it has failed due to something such as a connection error so the script outputs this and ends. Finally, if this is greater than 1... then something has gone very wrong and that shouldn't have happened.

Get User Items

```
//Check 'useritems' table for any entries that belong to the specified player.
$playerID = $_POST["playerID"];
$sqlGetInventory = "SELECT * FROM useritems WHERE playerID = '" . $playerID . "'";
$playerInventory = $conn->query($sqlGetInventory);
```

Firstly, a single user is taken as an input. This is the user whose inventory will be retrieved. It then takes this user's ID and checks for it in the 'playerID' column of the 'useritems' table. Any that match are items in that players inventory so are returned by the SQL statement.

```
//Format data to be used in unity: {id},{name},{data1},{data2}/{id},{name},{data1},{data2}/etc...
$inventoryString = "";
if($result->num_rows > 0){
    while($item = $playerInventory->fetch_assoc()){
        $inventoryString = $inventoryString . $item["id"] . "," . $item["name"] . "/";
    }
    die($inventoryString);
}else{
    die("Specified user has no items.");
}
//If it has 0 rows no items were found so the specified user has no items.
```

Next, the data is formatted so it can be sent and used easily. It is converted into a string which contains all the items separated by '/', with their data separated by ','. This ensures data is compatible with any system put into the game as it is just a string.

Get Item

```
//Get item data from database.
$itemID = $_POST["itemID"];
$sqlGetItem = "SELECT * FROM items WHERE id = '" . $itemID . "'";
$item = $conn->query($sqlGetItem);

//Turn item data to string to be used.
$itemData = "";
if($item->num_rows > 0){
    $item = $item->fetch_assoc();
    $itemData = $itemData . $item["name"] . "," . $item["type"]; //
}else{
    die("Item not found.")//Item with that ID doesn't exist.
}
```

This works very similarly to the get user items script except it only retrieves one single item, and it retrieves this from the 'items' table so gets all the data for that item, but only the non-unique data; it retrieves data that is the same across every copy of that item.

It works by taking an ID of an item as an input and tries to retrieve the row with this ID from the 'items' table. If it successfully retrieves data, this is formatted into a string with each data piece separated by a ','. If no data is found, it stops the script and outputs the error that the item wasn't found.

4.2.2 Database tables

Screenshots from the MySQL admin panel.

'users'

id	name	password	date	xp	clanid
2			2024-06-21	2000	0

'items'

id	name	type	speed	strength	distance	rarity
1	Placeholder	Miscellaneous	0	0	0	0

'useritems'

userid	itemid	quantity	level	date	slot
0	0	1	1	2024-12-03	inventory

4.3 Testing

4.3.1 Tests

id	Description	Inputs	Expected Outcome	Actual Outcome	Changes?
0.1	Login script	Blank inputs	"Check login"	"Check login"	No
0.2	Login script	Correct username, wrong password	"Check login"	"Check login"	No
0.3	Login script	Correct username and password	"Login success"	"Login success"	No
1.1	Create account	Existing username	"User with that name exists"	"User with that name exists"	No
1.2	Create account	Mismatched passwords	"Passwords must match"	"Passwords must match"	No
1.3	Create account	Blank/Short username/password	"Username and password must be longer than 3 characters"	"Username and password must be longer than 3 characters"	No

1.4	Create account	Matching passwords and new username	"Account created" + new entry in database with correct data	"Account created" + new entry in database with correct data	No
2.1	Get Item	"0" (a placeholder item has been set up with id 0)	"0,placeholder,junk,0,0,0,1"	"placeholder,junk"	Yes – Add missing fields to output.
2.2	Get Item	"100" (no item with id 100 currently)	"Item not found"	"Item not found"	No

4.3.2 Fixes

[2.1] Add missing fields.

It currently retrieves the correct item but does not return every data field only name and type. Extend the die() output to include every data point.

```
$itemData = "";
if($item->num_rows > 0){
    $item = $item->fetch_assoc();
    $itemData = $itemData . $item["name"] . " , " . $item["type"] . " , " . $item["speed"] . " , " . $item["strength"] . " , " . $item["range"] . " , " . $item["rarity"];
    die($itemData);
}
```

4.4 Sprint Overview & Maintenance

In this sprint the goal was to create a scalable database with the functionality to be interacted with by the game itself through the use of SQL and php scripts.

Storing the data itself was done using a MySQL database with 3 tables. These tables were planned out and prototyped to allow the database to be normalized. This means that it uses the smallest space possible to store the required data, while still being able to access any part of the database whenever it is needed.

The system for the game to interact with the database was created using a set of php scripts that each allow data about the player, the items, and the items the player has to be retrieved.

Finally, the format for the actual item data was decided in a way that allows all items to be stored together and allows the game to construct them from the data it is given.

There were a few small issues with outputting the correct data fields, but these were easily fixed and didn't cause any major problems.

The next step is allowing the player to access, use and obtain these items in-game.

5.0 Sprint 4 – Modular Items

5.1 Design

5.1.1 “Type Structure”

Each item in the game has a data field “type”. This defines what that item can do and which scripts should be called by it upon different usages.

Some possible types include “sword” “pistol” “automatic rifle” “health consumable” “gadget” etc. For purposes of creating a prototype item, the use function for the “pistol” and “health consumable” will be used.

For each type of item, it will have the three main stats (speed, strength, distance). Depending on the item type, each of these stats will do something different. For example, for weapon types such as a “sword”, the strength would relate to damage and distance to stamina, while for types such as “health consumable”, strength would relate to the amount of health it adds or removes, the speed to how fast you drink it, and distance to damage/heal over time if needed.

Every item will also have a use() function, that varies based on the type. Consumables would be eaten or drunk and effects applied to the target entity, weapons do an attack.

Attributes		
Name	Data	Why
Speed	Int	Core stat – tweaks the item behaviour
Strength	Int	Core stat – tweaks the item behaviour
Distance	Int	Core stat – tweaks the item behaviour
Type	String	Modifies the use and characteristics of the item.
Methods		
Name	Parameters	Description
Item	Int, int, int, string	Constructor. Returns a new instance of the item with the 3 core stats and type set as the inputs.
Use	Entity	Uses the type attribute to determine then call the correct use function. Takes an entity as a target.
Use Functions		
Health Consumable	Entity	Takes an entity as the target and modifies the target’s health by this item’s strength attribute. Can also apply over time based the distance attribute.
Pistol	Transform	Takes a transform as the input to know the players position and rotation so that it can calculate where the shot lands. Then damages the entity it hits (if it is an entity). Acts as a ranged weapon.

5.1.2 Use functions

All use functions should take one parameter: the target. For some this will be set as the user by default, and others could be set using ray casts or references.

Health Consumable

This type should have a use function that causes 1 count of the item to be removed from your inventory, and for the target entities health to be modified, either negatively (damage) or positive (health). It should also be able to cause a damage over time effect using the distance stat.

Stat	Meaning (+)	Meaning (0)	Meaning (-)
Speed	The speed the item takes to use.	Instant use.	Nothing (takes modulus).
Strength	The amount of instantaneous health to heal the player by.	Consumable has no effect.	The amount of instantaneous health to damage the player by.
Distance	Duration of damage over time effect. (Strength is damage/health per second).	No damage over time (instant damage/health).	Nothing (takes modulus).

```

Use(target):
    //Add item animations here later.
    Wait for <speed> seconds.
    if <distance> > 0:
        apply damage <strength> to target ever second for
        <distance> seconds.
    else:
        <target>.takeDamage(<strength>)
        //(+ = damage, - = heals).

```

This script aims to change the targets health by the value in its strength stat. It should do this in one of two ways; over time, or all at once.

If the attribute distance is greater than 0, this means that the method should be used over time. To do this a timer will be created, and the target will be damaged/healed by the amount in the strength attribute, once per second until the timer reaches zero.

If the attribute distance is 0 or less, this means the instant method should be used. This will use the same function to apply the heal/damage that the over-time method uses, except it will only be used once with the value in the strength attribute. Additionally, there will be no timer as the effect should be instant.

Finally, though this is a health potion, it could be used for poison potions, damage potions, heal potions, or slow heal potions. This is because it affects health but doesn't

have to only take it away or add it but can do one or the other, creating the dual functionality.

Pistol

This type's use function should simulate the effect of gun. The use function should keep track of an ammo count, and when used should create a raycast and damage whatever it collides with, if that thing can be damaged.

Speed	Rate of fire. (How many times per minute it does the shoot effect).	The weapon cannot be used.	Laser.
Strength	How much damage each individual shot does.	Does no damage (could be used for funny stuff like a nerf gun).	How much health is restored to the target per shot. (Healing gun).
Distance	The maximum range of the raycast.	Fires blanks. A little useless but it is an unintended functionality that would appear.	The same as if it is positive. (the modulus will be taken).

Use()

```
//Add item animations here later.
do a raycast directly forwards, from screen or gun
return raycast hit as <target>
<target>.health - <damage> (- is health, + is damage)
For testing:
    write raycast hit name
    write health of object after modification if applicable.
```

Pistol types (and other ranged items) should not have a target parameter like consumables. Instead, they find their own target by casting a ray directly forwards. This could be from either the centre of the camera, or the end of the physical gun model, however this will have to be prototyped to see which is more practical to play with.

As every item will have an entity class, which will be covered later, this script will make use of that by modifying the target entities health value, if it is an entity, by this item's damage stat. This will create the effect of the hit entity either taking damage or healing depending on the item.

The raycast itself should be modified using the distance stat, that will put a limit on the maximum range of the raycast. As negative range would be backwards which is useless, a modulus of the range should be taken so it is treated as if it was positive.

Finally, the rate of fire of the item should be regulated by starting a timer every time the use function is called, that prevents it from being called a second time until the timer has reached zero.

5.1.3 3D model swaps

The 3D model of the equipped item should change based on the type value. For now it will use a generic model for each type. In the future it could be changed to base the model off the name of the item and have unique models for each item.

To do this, the type value will have a set method that causes a model swapping procedure to be called every time the type value is modified. This procedure should then load the 3D model with the same name as the type value.

To store and be able to load these models, unity's resources folder will be used. This is a folder that is included in the final build, allowing the contents to be accessed at runtime, and then instantiated in the scene. This is the same folder that was used in the level generation scripts.

5.1.4 What to test

Test	Expected Outcome
Pistol Use – Point at entity	Entity takes damage equal to item strength value.
Pistol Use – Point at nothing	No errors, game should continue with nothing happening.
Pistol Use	Shooting animation
Health Consumable Use	Increase the user's health by strength value
Health Consumable Over Time	Increase the user's health by strength value every second for distance value seconds.
Swap type value	The use function called should change, and the 3D model the player holds should also change.

5.2 Development

5.2.1 Use Functions

Base Use Function

```
public void Use(Transform Origin, Entity Target = null){
    switch (type){
        case "health consumable": HealthConsumableUse(Target); break;
        case "pistol": PistolUse(Origin); break;
        default: Debug.Log("Generic type."); break;
    }
}
```

This takes a transform and an entity as an input. These are taken so they can be passed through into the correct use function.

Firstly, this function compares this item's type attribute to the names of the implemented use functions. If it matches one, then it calls that function, entering the

parameters from itself into this new function. If the type doesn't match any implemented string, it simply does nothing, as this item isn't intended to be used.

The reason functions are called like this and not directly, is because it allows an item's type to be set easily without having to find and copy all the correct code for it. It is also modular as each use function is separate, meaning new items can be easily implemented and maintained.

Calling the functions

When attached to entities, use functions will be called by other scripts that are running, however when on the player, nothing calls it. Therefore, it must be manually called by adding an if statement inside the Update function of the player movement script. This checks for a mouse click, and if it is present, it calls the use function of the player entity's equipped item.

It is done this way, in a separate script, as this allows for the inputs to be independent from the item, making it very simple to change or add items or inputs.

```
if (Input.GetButtonDown("Primary Fire")) { player.EquippedItem.Use(this.transform); }
if (Input.GetButtonDown("Secondary Fire")) { player.EquippedItem.Use(this.transform); }
```

To do this, Unity's input API is used that allows for multiple physical keys to be referenced under one name. In this case "Primary Fire" refers to the mouse left click. This system is used because if inputs need to be modified, the physical button can be swapped in a GUI instead of manually changing every reference in code.

Health Consumable

```
#region Use Functions
1 reference
private void HealthConsumableUse(Entity Target = null){
    UseDelayTimer = this.speed;
    if(distance > 0){ Target.TakeDamage(this.strength, this.distance); }
    else { Target.TakeDamage(this.strength); } //Uses Entity's take damage function.
}
```

This is very small as it mostly makes use of the already implemented damage function that the entity class has. For the damage over time, it simply passes the distance attribute into the function as the parameter for time. For both damage over time and instant damage, it passes the strength attribute as the damage per second, or damage to apply, respectively.

It uses the damage function of the entity passed in the parameter "target" as this allows it to either apply the effect to itself or others.

Finally, upon use, this item creates a timer that prevents it from being used again until after a certain time. This is because it can prevent it from being used over and over,

many times in a short amount of time, which would seem very unnatural and jarring to the player. The short delay prevents this.

Pistol

```
private void PistolUse(Transform Origin){
    if (UseDelayTimer != 0){//Allows setting a fire-rate for pistols.
        RaycastHit hit;
        trace = new Ray(Origin.position, Vector3.forward, out hit, this.distance);
        Physics.Raycast(trace);
        try{ hit.transform.GetComponent<Entity>().TakeDamage(this.strength); Debug.Log("Hit " + hit.transform.name);}
        catch{ Debug.Log("Nothing Hit"); }
        Debug.Log("Pistol shot");
        UseDelayTimer = 1/(speed/60); //Speed = rpm, speed/60 = frequency(Hz), 1/f = T(s) = delayTimer
    }
}
```

Firstly, the function checks if the delay timer is greater than zero. If it is, the pistol is not ready yet so the function ends and nothing happens. However, if it is not zero, the main function is run. Firstly, it creates a raycast going directly forwards out from the origin. This means that it will come out of the front and centre of whatever object the transform refers to. For example, when attached to the player, this should cause it to create the ray coming out from the center of the screen and directly forwards, which is a behaviour commonly seen in games with ranged weapons to simulate the projectile going where you aim.

Next the object it hit is returned. The script then tries to access the entity class of the script. If this fails, it means the pistol hit nothing and nothing should happen. However, if this succeeds, it means that something was hit, so the hit entity's damage function is used to apply damage equal to the strength attribute of the item.

Finally, to create the rate of fire effect seen in many games, a timer is applied at the end of the function. This is based on the speed attribute of the item, and the item cannot be used again until the timer reaches zero.

5.2.2 3D Model Swaps

Resources Folder

An additional folder is added next to the folder for level generation resources. This folder contains the models. For the purpose of testing this function, weapons from a free asset pack will be used. The pack can be found in the **Total requirements met: 12/16**

7.4 Summary

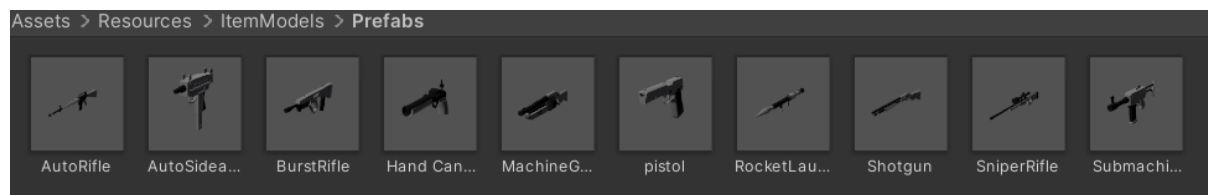
12 out of 16 requirements for the project were met which is a 75% success rate. The other components have either not been started or have had the framework laid out for them allowing them to be completed with relative simplicity in the future. This shows that the systems planned to be implemented were implemented and work as desired.

The project itself was designed to be a rogue like shooter with a database integration, and the project at this stage has set out a solid basis to build upon. All the main systems

such as player movement, items and their interactions, enemies, and most of the server functionality have been implemented. This allows each of these components to be used and put together to create a full game experience that people can enjoy.

In conclusion, while not providing a full playable game yet, the project has laid out a framework for the game to be built upon, with minimal more code interaction required, simply using the created systems.

Bibliography.



This allows the script to access these models using by loading them at runtime through the path “./ItemModels/Prefabs/<item-name>”, simplifying the script greatly.

Model Swap Script

```
public void ModelSwap(GameObject Hand){
    string modelName = this.transform.gameObject.name.ToLower();
    modelName = "ItemModels/Prefabs/" + modelName;
    //Get path to resources folder.
    //Models in folder must be lower case named.
    Debug.Log("[Item] Model path: " + modelName);
    UnityEngine.Object model = Resources.Load(modelName);
    Instantiate(model, Hand.transform);
    //Load and create item in scene.
}
```

This takes a unity Game Object as an input, as this allows the model created at the end to be assigned as a child of another object. In this case it will be assigned as a child of the player’s right hand, which will

cause the item to appear at that position, and track the hand when the player moves.

To access the model, it first takes the name of the object this script is assigned to. In the plan it was going to take the type of the object instead, however taking the name allows different objects of the same type to have unique models, creating more motivation to collect them. Then this name is added to the end of the path “itemmodels/prefabs” which returns the model with that name for the script to use.

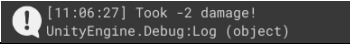





Finally, the Instantiate method is used to create the loaded item in the scene, with the player’s hand as the parent object. This should cause the correct model to appear in the player’s hand whenever the script is called.

This script will be called whenever the player is created, and also whenever the item it is using is updated, because this maintains an accurate visual representation for the user.

5.3 Testing

5.3.1 Tests

i d	Descriptio n	Inputs	Expected Outcome	Actual Outcome	Changes ?
0	Pistol – point at nothing	Left Mouse	Debug console says, “nothing hit”.	Debug console says, “Player hit”.	Yes – Ignore player.
1	Pistol – Raycast comes out the camera	Left Mouse	Draws a ray from the centre of the player’s face (the camera).	Ray goes from 0,0,0 in a forwards direction.	Yes – Change origin and direction.
2	Health – Use to heal	Right Mouse	Player.TakeDamage should be called	Method called, adds 2 health. Health doesn’t	Yes – Redo test

		+ strength = -2	and health increase by 2	increase because its at it maximum already. 	when not at max health.
3	Health – Use to heal (repeat)	Same as last test but damage player first	Player Health increases by 2.	Player health increases by 2 as well as the function being called.	No.
4	Health – heal over time	Right Mouse + strength = -1, distance = 2	Player health should increase at a rate of 1 per second, for 5 seconds. (ends 5 higher than it starts).	Function is called for correct amount of time (2 seconds) and increases the health by 1 each second. The results is slightly off 5 due to frame timings, however the amount is negligible.  (Started at 3).	No.
5	Health – Use to take damage	Right Mouse + strength = 2	Player health should decrease by 2 instantly.	 (Started at 10).	No.
6	Health – Damage over time	Right Mouse + strength = 1, distance = 5	Player should lose 1 health per second for 5 seconds.	 (Started at 8). This is acceptable because it rounds to 3 and the difference between them is negligible.	No.
7	Models - pistol	Name = Pistol	A model of a pistol should appear in the player's hand.		No.
8	Models – health consumable	Name = health consumable	A sphere should appear in the player's hand (temporary model).		No.
9	Model - exception	Name = random characters	Default item should appear in player's hand (cube).	Nothing happens.	Yes – add default.

5.3.2 Fixes

[0] Ignore Player

The raycast is hitting the player, because it comes from the player's position. This means that it starts inside the player. Therefore, as it goes outwards, it collides with the player and returns it. This isn't the intended behaviour, so a layermask is added to the raycast that causes it to collide with everything except the player.

```
Physics.Raycast(Origin.position, Vectro3.Forward, out hit, this.distance, ignorePlayer);
```

In unity every object can be assigned a layer, and a layermask is a way of referencing those layers. Adding a parameter to the raycast to ignore anything in the layer “player”, prevents it from colliding and returning any object from that layer.

[1] Change Origin and Direction

```
Physics.Raycast(Origin.position, Camera.main.transform.forward,
```

The origin was set correctly, however the direction was originally set to Vector3.Forward, which refers to north. Instead, camera.main.transform.forward is used, which takes the facing direction of the main camera (the player camera).

[9] Add default model

```
try {
    UnityEngine.Object model = Resources.Load(modelName);
}
catch {
    UnityEngine.Object model = Resources.Load("ItemModels/Prefabs/default");
    //Revert to default if there is no model.
}
Instantiate(model, Hand.transform);
//Load item in scene.
```

The load method is moved into a try function that means if it fails and throws an error, the script instead runs the code in the catch function.

This means that if no model is found with the current item’s name, then it will load and create the default model, a cube.

5.4 Sprint Overview & Maintenance

In this sprint the goal was to create a template for all future items that could be implemented. It was also to create two initial items, one that affects things it hits, and another that affects the user. Finally, it was also intended to allow items to be saved to and loaded from the server, and to allow items to be represented by models in-game.

The template for items, along with the initial first two items, was created effectively. The methods remain reusable and modular, allowing them to be used as templates as intended. This should make future maintenance such as creating or modifying items much easier as only a few parameters must be set.

The two created items both function as intended, however could use a visual representation such as animations. This is because it is currently hard to tell they are doing anything unless you destroy another object, unless you look at the debug console, which most people never will. This could be added later by simply calling animations in existing methods.

For the server functions, these were not implemented in this sprint as it became too crowded, so should be dedicated its own sprint. However, the process should be simple, as it can simply fill in the created templates using data from the server, making the future maintenance and addition of this feature theoretically quite simple.

Finally, for switching item models, this works effectively and allows new models to be created and assigned without modifying any further code, making maintenance and

addition of new models extremely easy. Furthermore, it also handles if no model is added, meaning it is robust and able to handle exceptions or errors.

6.0 Sprint 5 – Enemies (angry cubes)

This sprint aims to make functioning enemies and targets. It will create a script that can be attached to any Game Object to give it health, the ability to be damaged and destroyed, and will also create a script that will control enemy “AI” movement.

6.1 Design

The enemies will be split into two components: the “Entity” script and the “Enemy Movement” script. The entity script will handle health, being damaged, and any resistance to damage or effects applied to the enemy. The Enemy Movement script will handle the enemy’s attacks, movement and states.

6.1.1 Entity Class

This class should be a single script that acts on its own. It should keep track of entity data such as its name, health and stats. It should be able to be attached to any item that will have a health pool or any abilities, or anything that can be destroyed. It should also have methods for taking damage, using the equipped item.

Attributes				
Name	Get	Set	Type	Description
Name	Public	Private	String	Entity name
Health	Public	Protected	Float	Remaining health of the enemy.
Speed	Public	Protected	Float	Movement speed
Strength	Public	Protected	Float	Melee attack damage
Distance	Public	Protected	Float	Multiplier on the detection range.
Methods				
Name	Access	Arguments	Description	
Take Damage	public	Float, float = 0	Negative = heal, positive = damage. Should handle checking if entity is alive after damage. Could also call/trigger animations. Boolean to apply damage over time (DOT) if needed.	
Destroy Self	private	-	Called when health pool reaches 0. Allows for death animations/events to be added.	
Use equipped item	public	Item, Entity	Use the referenced item (probably equipped item) with the referenced entity as a target (probably the player).	

Take Damage

This should be a public method that any other script can call. It should take a number as an argument, and a Boolean. The number is how much damage this enemy should take, while the Boolean states whether this is a damage over time effect.

This should simply subtract the damage value from the entity's health pool. Because it is a subtraction, inputting a negative number will cause it to increase its health, allowing for healing to be created simultaneously.

For validation of this method, when making a change to health it should check the result. If it is above the entity's maximum health, it should set the entity health to its maximum instead of simply increasing it by the amount stated. If the value is less than zero, it should call this entity's destroy method.

Finally, the second parameter is optional, however is also an int and can create a damage over time effect. This will be used as a timer and decreased by 1 every second. While the number is greater than zero, the damage function should be repeatedly called with the damage amount. Conversely, this can again be used as a healing effect, to create a heal over time effect.

Destroy Self

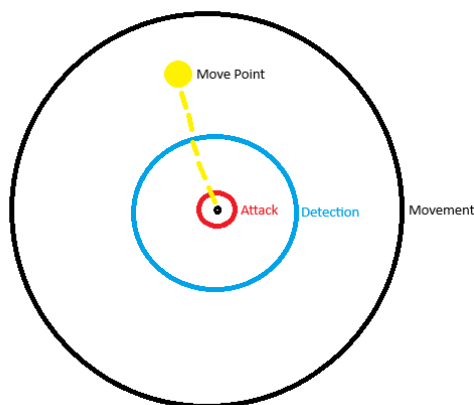
This method will be very simple and will be called by the take damage function when health hits zero. It will simply disable the unity object which makes it disappear. The reason this is split into a function is to allow death or destruction animations and events easier to implement in the future if needed.

Use Equipped Item

This script should contain a serialized reference to the entity's equipped item. This means that the variable is not visible to other unity scripts, but can be changed in the unity editor, making the process of creating new enemies/characters easier.

The function itself should call the use function of that item. This function only acts as an intermediary between the enemy/player script, and whatever item they have equipped. It means that an equipped item can be changed without changing every reference to it in those scripts, and only once in this script instead.

6.1.2 Enemy States



The player will have three states: patrolling, chasing and attacking. These will depend on the player's proximity and the distances will be calculated using ray casts. Each state will have a different movement and action that the enemy does to make them seem more alive by not simply always going directly towards the player. It should also aim to prevent the enemies from becoming stuck on level features such as buildings or decorations.

Patrolling State

This state is used when the player is out of both attacking and detection range of the player. It will be the default state and when active will cause the enemy to wander around its assigned area randomly.

```
int moveSpeed
int moveRange
Vector3 movePoint

unity Update function(){
    if(movePoint is unset){
        while(movePoint is unset){
            pick a random point within radius moveRange of entity.
            spherical raycast around the point
            if(raycast doesn't collide with any obstacles){
                movePoint = that point
            }
        }
    }
    look at movePoint.
    cast ray forwards by a short range for example 3m.
    if (ray collides with pathfinding obstacles){
        move forwards by move speed.
    }else{
        unset movepoint so a new point can be calculated
    }
}
```

For this state, the enemy will have a movement speed variable, a variable to define the movement range and the move point.

Every frame, the enemy checks to see if it currently has a point to move towards. If it does have this point it should point towards it and then move forwards. However, if it moved straight forward it could collide with or go through another object. To prevent this, it should cast a linear ray by a short distance directly in front of it by a short distance. If this collides with any obstacles, the current path is invalid so it should unset the move point so a new one can be set.

If no move point is set, it should find one. To do this, a random point around the enemy should be chosen that is within the range specified, i.e: a circle with a radius of the movement range. Once the point is chosen, it should check to see if it inside anything. To do this a raycast could be used. If it is inside something a new point should be chosen. Otherwise, the point is valid and should be set as the new move point.

Chasing State

In this state, no pathfinding algorithms or obstacle detection is needed. This is because the enemy should only chase the player if they have a direct line of sight. Therefore, if the enemy is chasing the player, there are no obstacles in the way, so no obstacle detection or pathfinding is needed.

This state will activate once a player is within range of the enemy and the enemy has a direct line of sight to it. This could be checked using raycasts, i.e: a linear one to check line of sight, and a spherical one to check if it is in range.

```

each frame:
if (player in detection range){
    check if enemy has a direct line of sight to player.
    if it does{
        set player position as the last known player position
        move towards the player
    }else{
        move to the players last known position at the chase speed
    }
}

```

Each frame the enemy should check if there are any players within the specified detection range. If there are, then it should check if it has a line of sight. This could be done by looking at the players direction and doing a raycast and seeing if it hits the player or not. If this check is successful, the enemy should then set the players current position as its last known position and move towards it.

The reason it moves towards the player's last known position and not the player's actual position is because if the player hides behind something the enemy will still have its last known position so can keep moving towards that. If it didn't have the player's last position, it would either have to stop moving towards the player or keep moving towards it as if it had X-Ray vision.

Attacking State

```

when player within melee range:
    loop:
        stand still
        get item that it is holding
        use item with player as target
    repeat

```

This state should activate whenever the player is within one to two metres of the enemy. In this state the enemy should stand still and use any close-range items in its

hand such as a sword or a bottle of water.

For example, attacking enemies will most likely have a weapon and friendly "enemies" could also be created that hold something like a healing potion and heal the player. This allows for allies or pets to be created using the same script by giving them a huge detection range and a helpful item like a healing potion, so they follow the player and help them.

6.1.3 Key Variables

Name	Type	What For
MoveSpeed	float	Speed of the enemy when patrolling
MovePoint	Vector3	Position for the enemy to move to (changes when it reaches it)
MoveRange	float	Distance away from the enemy a new move point can be
MovePointSet	bool	Determines whether the move point needs to be assigned or not
ChaseSpeed	float	Speed of the enemy when chasing
DetectionRange	float	How far the player must be from the enemy for it to start chasing
LineOfSight	bool	Whether the enemy has a line of sight to the player
PLastPosition	Vector3	The last position of the player and where the enemy chases

EquippedItem	Item	The enemy's item that it will use when attacking
AttackRange	float	How far the player must be from the enemy to start attacking

6.1.4 What to test

Test	Expected outcome
Move within enemy detection range.	Enemy moves towards player when it is in line of sight, then moves towards the players last seen position when it loses line of sight.
Move within enemy attack range.	Enemy stands still and attacks player over time using its equipped weapon.
Move outside of enemy range.	Enemy wanders around randomly, not getting stuck on anything and staying on the ground.
Attack enemy	Enemy health decreases and any script in TakeDamage() is executed.
Heal enemy	Enemy health increases and any script in TakeDamage() is executed.
Change enemy speed, strength or distance.	Enemy movement speed / melee damage / detection range increase.
Enemy try to go through obstacle	It should calculate a new walk point and start moving towards that instead, before it hits or walks through the object. This should only happen with objects in the correct layer.

6.2 Development

6.2.1 Entity Script

Take Damage

```
public void TakeDamage(float amount, float time = 0){
    if (time > 0){
        dotTimer = time;
        dotDamage = amount;
        Debug.Log("Applied DOT");
    } //Initiate DOT
    else {
        health -= amount;
        Debug.Log("Took " + amount + " damage!");
    }

    //Check health state
    if (health >= maxHealth){
        health = maxHealth;
    } else if (health <= 0){
        DestroySelf();
    }
}
```

Firstly, it takes the amount of damage and if it is a damage over time effect or not as parameters. The procedure then checks what kind of damage effect this is by checking if the time variable is greater than 0. If it is, it means this damage should be applied over time, and if it is 0, then it should be applied in one chunk.

If it is instant damage, the script simply subtracts the amount of damage from its health attribute. If it is damage over time, it initiates the damage over time effect by setting the temporary variables dot timer and dot

damage to the time and amount parameters.

Finally at the end of the procedure it checks if the health is within a suitable range and resets it to max if it exceeds the range and kills the entity when the health goes below zero by calling the destroy self procedure.


```

public void Update(){
    if (dotTimer > 0){
        TakeDamage(dotDamage * Time.deltaTime);
        Debug.Log("Taking DOT: " + dotDamage + " --> " + this.health);
        dotTimer -= Time.deltaTime;
        //DOT independant of frame rate so ppl with 1000fps dont just phase out of existence.
    }
}

```

To implement the damage over time (dot), each frame the script runs one function to check if the dot timer is greater than zero. If it is, it means that dot needs to be applied. To do this it calls the take damage function using the dot damage attribute as the damage, however this is multiplied by the time since the last frame to make it independent of frame rate. Finally, to make the dot have a timer, the attribute dot timer is decremented by the time since last frame, so it lasts the number of seconds specified.

Destroy Self

This script has little function currently and only serves as a clean way to allow death animations or events to implemented in the animations sprint if they are needed.

```

private void DestroySelf(){
    Debug.Log("Health at 0... Destroyed.");
    //Any animations and stuff for when dying here
    gameObject.SetActive(false);
}

```

It calls the unity function “set active” of the game object the script is attached to, with a parameter of false. This causes the object in the unity to be

deactivated meaning it disappears from the scene and no longer interacts with anything, and any associated scripts are ended.

Use Item

This script is used to connect the item the entity is using to the entity. It allows for an entity to be told to use its item, and the entity script works out and finds the correct item.

```

private void UseEquippedItem(){
    EquippedItem.Use(this.transform, this);
}

```

To do this it has a reference to the item equipped, allowing different instances of entities to have different

items. The item’s main use function is then called, with the entity’s position as the origin. It also sets itself as the target because no item types are currently implemented that require a target other than itself.

6.2.2 Enemy Script

Patrolling State

The very first step to this script is to check for collisions and change its move point if there are any.

```
private bool CheckObstaclesInFront(float range){
    Quaternion rotation = transform.rotation;
    transform.LookAt(playerLastPosition);
    bool result = Physics.Raycast(transform.position, transform.rotation * Vector3.forward, range, obstacleLayer);
    transform.rotation = rotation;
    return result;
} // Check for obstacles in obstacle layer in front of self by specified range.
```

To do this, it uses a procedure that is called at the start of every patrolling method. First, it rotates the player temporarily towards its move point and shoots out a short ray directly in front. If this collides with anything in the obstacle layer in unity, such as terrain or walls, the procedure returns true. If it doesn't collide with any obstacles then it returns false.

```
private void Patrolling(){
    if(CheckObstaclesInFront(3.0f)){
        Debug.Log("Obstacle; recalculating route.");
        movePointSet = false;
    } // Check for obstacles in front of it. Not needed when chasing as it only chases with direct line of sight and not needed when attacking as it

    if(Vector3.Distance(transform.position, movePoint) < 1f){
        movePointSet = false;
    } // Unset point to find a new walk point when within a certain range of it.

    if(!movePointSet){
        SetWalkPoint();
    }

    transform.LookAt(movePoint);
    transform.position = Vector3.MoveTowards(transform.position, new Vector3(movePoint.x, groundLevel, movePoint.z), moveSpeed * Time.deltaTime);
    // Move
}
```

Using this procedure's output, the main patrolling method runs. If the output was true, then it means the current route for the enemy is invalid, so it changes the bool "movePointSet" to false. If it returns false, the method continues as normal. Next, it checks if the distance to the move point is less than 1 metre. This is so that it can select a new move point when it reaches the current one, allowing it to continue patrolling.

The validation of the current move point (if any) is now complete, so it moves on to the next step; either calculating a new point or moving to that point. If at this point the move point is unset for any reason, for example: if the enemy has just been created, one of the conditions weren't met, or it has just exited from another state, it calls a new procedure to set the move point.

```

while(!movePointSet){
    float randomX = Random.Range((transform.position.x - moveRange), (transform.position.x + moveRange)); //Gen Random X
    float randomZ = Random.Range((transform.position.z - moveRange), (transform.position.z + moveRange)); //Gen Random Z
    RaycastHit hit;
    point = new Vector3(randomX, transform.position.y + 10f, randomZ);
    Physics.Raycast(point, Vector3.down, out hit, Mathf.Infinity, walkableLayer); //Find Y Level of AI walkable ground
    point = new Vector3(point.x, hit.transform.position.y + 0.1f, point.z);

    if(!Physics.CheckSphere(point, 1f, obstacleLayer) && Physics.CheckSphere(point, 1f, walkableLayer)){
        movePoint = point;
        movePointSet = true;
    } //Check if point inside obstacle
}

```

This function is contained inside a loop, because not all positions are valid, and it should continue until a valid position is chosen. The procedure itself chooses a random position in 2d space, then moves this point to 10 metres above the player. This allows it to account for any elevation changes in terrain. It then executes a ray cast directly downwards and finds the first object that is walkable below it. The y position of the chosen point is then replaced with the height of this object's surface, meaning the move point has been set to a point on the floor.

This point is then validated. To do this it casts a spherecast to return all objects within a radius of 1 metre around the point that are obstacles. If any are returned, the point is invalid and nothing happens, which causes the loop to restart. If nothing is returned, the point is valid and not inside or near any obstructions so the "movePointSet" bool is set to true and the move point is set to the randomly chosen and validated point.

```

transform.LookAt(movePoint);
transform.position = Vector3.MoveTowards(transform.position, new Vector3(movePoint.x, groundLevel, movePoint.z), moveSpeed * Time.deltaTime);
//Move

```

Finally, as the last step of the patrolling method, the enemy moves towards the current point by using unity's move towards function. The way this works is by taking two points in space, the first being the current position and the second the target, and subtracting their locations to get a distance between them. This distance is then added on to the original position, but in small increments with equal magnitude to the speed parameter. This causes the enemy to move directly towards the move point at the specified speed.

As the Patrolling procedure is run in the Update method, it is called every frame, meaning each frame it checks for obstacles and the validation of the current movement and chooses a new point if necessary, then moves to that point.

Chasing State

```
private void Chasing(){
    Quaternion rotation = transform.rotation;
    transform.LookAt(player);
    RaycastHit hit;
    bool lineOfSight = Physics.Raycast(transform.position, transform.rotation * Vector3.forward, out hit, Mathf.Infinity);
    if(hit.transform.gameObject.layer == playerLayer){
        playerLastPosition = player.position;
    }
    transform.position = Vector3.MoveTowards(transform.position, playerLastPosition, chaseSpeed * Time.deltaTime);
}
```

First this needs to find out if the enemy has a line of sight to the player. To do this, it rotates the enemy to be looking directly at the player (whose position is pulled from a public variable in the player script), and the enemy shoots a ray cast. This ray cast will then either hit the player or something else, so it checks if the hit object is a player. If it is, it sets the players last known position to the player's current position. If it the ray cast hits something else other than a player, it doesn't update the players last known position.

It then moves towards the player's last known position. This creates the effect of if the player goes behind a wall and out of sight, the enemy will continue moving to the last place it saw the player.

Attacking State

```
private void Attacking(){
    Debug.Log("Attacking");
    gameObject.GetComponent<Entity>().EquippedItem.Use(this.transform, gameObject.GetComponent<Entity>());
}
```

This function is activated whenever a player comes within range of the enemy. The function itself makes use of existing methods to simplify it into one statement. It uses a reference to the item this entity is holding, then calls the use function of that item. The justification of doing it like this and not hard coding an attack, is because you can simply modify what the enemy has equipped without touching any code, and this will modify the enemy's actions, making it unique. It also allows for simple maintenance when creating new levels as an enemy can simply be added, given an item, and they will interact seamlessly.

Choosing states

```
try { state = (playerDistance <= detectionRange) ? (playerDistance <= equippedItem.distance) ? 2 : 1 : 0; }
catch { state = (playerDistance <= detectionRange) ? 1 : 0; }

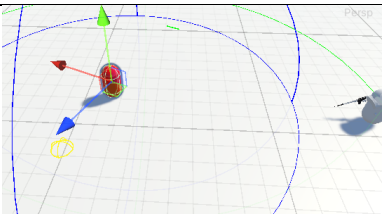
switch (state){
    case 0: Patrolling(); break;
    case 1: Chasing(); break;
    case 2: Attacking(); break;
}
```

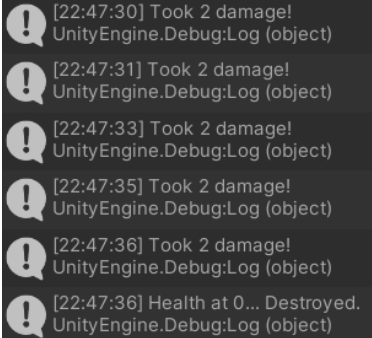
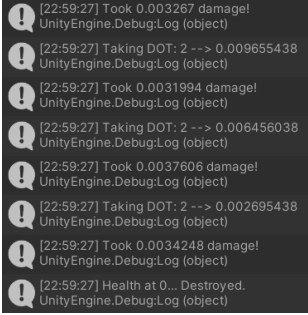
This is done by setting a state variable to one of 3 numbers. 0 for patrolling, 1 for chasing, and 2 for attacking. The setter for the variable sets it to patrolling if the player is within detection range, and attacking if it is also in the range of the equipped item. The setter is contained in a try statement so that if there is no equipped weapon, instead of throwing an error, it runs an alternate setter that only sets between patrolling and attacking.

Then, a switch statement is used to call the corresponding method for each state. This is all run every frame to ensure the enemy is always doing the correct action.

6.3 Testing

6.3.1 Tests

id	Description	Inputs	Expected Outcome	Actual Outcome	Changes?
0	Detected by enemy	Player < 10m away from enemy	Enemy should start moving towards you at the chasing movement speed.	Enemy starts moving towards the point (0,0,0) when you get within range.	Yes – Move to player not centre.
1	Enemy wandering	Player > 10m away from enemy	Enemy should pick a valid move point and move towards it at the specified movement speed.	 <p>Yellow – move point Blue – Detection range Blue arrow – movement direction</p>	No
2	Enemy attacking	Player < item range away (5m), item = pistol	Enemy should stop moving, and use the item on the player. (Player health decreases by 2 every few moments.	Enemy is stationary. Player health drops slowly in increments of 2.	No.

3	Entity instant damage	Take damage with argument of 2 & start hp at 10.	Enemy should take 2 damage. To add another test, repeating this 5 times should destroy the object.		No
4	Entity DOT	Take damage with argument of 2, 5 & start hp at 10.	Enemy should take 2 damage every 5 seconds. In practice this will cause the entity to destroy after 5 seconds.		No
5	Entity Use Item	Move player within 5m of enemy.	Entity should stop moving and player should take damage.	Error: no object referenced.	Yes - make enemy aim at player.

6.3.2 Fixes

[0] Enemy chasing target

The test demonstrated that the movement of the enemy was not being set properly as it was remaining the default value for a Vector3; (0,0,0).

```
if(hit.transform.name == player.name){
    playerLastPosition = player.transform.position;
```

When setting the playerLastPosition value, which is the one the enemy moves towards, it was previously comparing the hit object's layer. This value cannot be retrieved from a raycast so was returning false every time, even when hitting a player. This caused playerLastPosition to result to its default: (0,0,0). To fix this it compares the name of the hit object to the name of the object in the player reference.

Now the enemy chases the player, not the middle of the environment.

[5] Make enemy aim at player

```
RaycastHit hit;
Physics.CheckSphere(transform.position, EquippedItem.distance, playerLayer,
Entity tempPlayer = hit.transform.gameObject.GetComponent<Entity>();
transform.LookAt(tempPlayer.position); //Aim at player
```

To target the player, it first needs a reference to the player. To obtain this automatically, a

raycast will be used. If we assume that there is a player within range due to this function being called, a sphere cast can be used to return any player entity within range. This returned entity can then be taken to be the player, so the enemy is rotated to face it.

This section of code is added before the item use function is called, meaning that when the pistol or any other ranged item is processed, it will correctly be aiming at the player.

6.4 Sprint Overview & Maintenance

In this sprint, the goal was to create two scripts that can be added to any object to turn it into either a functioning enemy with pathing and attack states, or an entity with stats and health that can also be destroyed, or both.

The enemy script was originally designed to have 3 states: patrolling, chasing, and attacking. Each state works as intended, with the enemy able to follow the player when it is in sight, attack the player, and to move around on its own when it is idle.

Furthermore, an unexpected feature that arose was enemies can be locked to certain areas by setting only that piece of floor with an AI walkable tag. This means they will always try to stay in that region or return to it, unless they are chasing a player.

The entity script was also effective as it was supposed to allow it to be assigned to any object and allow it to be destroyed. At first there were some issues with other scripts identifying and making use of entity scripts, however this was solved. Now if you attach the script to any game Object inside of unity, that object can act as something with health and the ability to use an item (if one is given). For example, they can be applied to objects as targets, or enemies and players to give them health, stats, and the ability to use items.

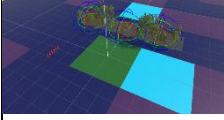
Finally, as for maintenance, these two scripts are independent from anything else, meaning that they shouldn't need any further modifications or testing. This also means that they can be applied to any item without conflicting with other components, which should make the maintenance of anything with these scripts very simple to expand.

In conclusion, this sprint has achieved the original goals it aimed to meet and has also allowed the created components to be easily maintained and expanded upon in any future development.

7.0 Evaluation

7.1 Final Tests

These tests will be chosen and to test the overall robustness of the project by testing features' general usability, and edge cases for user interaction.

Test Description	Inputs	Outcome	Good or bad
Level Generation			
Generate a tile set around the center of test level.	Add generator script to test level's floor object.		This works correctly as all the tiles generate as normal around the floor tile.
Database			
Login with random usernames + passwords combinations	Cabbage + 10234 Fluffy + turt13 Nothing + nothing	"Check login"	This is good as it means you can't login to non-existent accounts.
Check if login function can be overloaded	Spam login button very fast.	Slows down and eventually shows "too many requests".	This is good because it prevents the program from crashing by disconnecting the user. This is a feature of apache inside of xampp, the program used to run the database.
Login to account	Enter correct username + password	The data for that user is filled in to the player class.	This is correct as the filled in information matches the information for that user in the database.
Player			
Check if weapon function can be overloaded	Spam fire button.	Pistol shoots in time with specified rpm otherwise "[Pistol] Not ready"	This is good as it limits the fire rate of the weapon to preset values, allowing future weapons to feel unique. It is also good as it does not crash from too many inputs.
See if movement feels natural + easy to use.	Create and navigate a jumping puzzle.		
Enemies			
Get chased by enemy then escape it.	Walk near to enemy Walk away		
Die to enemy.	Stand by enemy		
Entities			
Assign entity script to random objects like trees, see if they can be	+ Random object + Entity script + Deal damage		

destroyed by weapons.			
-----------------------	--	--	--

7.2 Maintenance

For the future maintenance of the project, there are 4 main areas of focus. These are the creation and modification of items, creation of new level sets for generation, creation of enemies and damageable objects, and the management of users and their data.

In terms of items, their maintenance will be extremely minimal, and mainly focused upon the expansion of how many items are available. To create new items, simply add the item script to any unity game object and fill in the desired statistics. The item should also be given a name and a 3D model which should be placed into the resources folder with the same name as the item. To change the root behaviour of the item and not just the stats, the type must be changed. To do this simply enter in a different type. If that type doesn't already exist, then one function should be created, detailing what that new type does. In conclusion, the only new things needed for a new item are a 3D model of it, and possibly one function detailing what it does. This means that addition of new items, and changing old items, should be simple and manageable.

For the creation of new level sets, no code needs to be touched at all which creates an extremely developer-friendly workflow to add new sets. First you need 3D models for each tile you want to generate. At least one standard tile, one special tile, and one edge tile are needed. More can be added however they are not needed. You also need a text file detailing where special tiles can spawn. Simply write the number of the special tile in the corresponding character number in the file. Once this is done and put in a folder within resources, you can add a generation script to any object and give it the name of the folder, and a random layout will be generated around it using the set created.

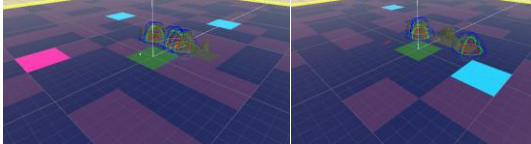
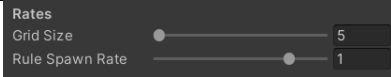
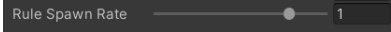
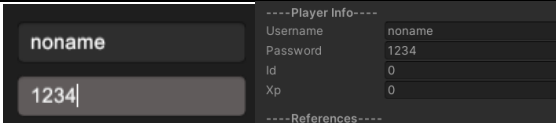

Next, creating enemies and damageable objects requires the use of 2 scripts, however they do not have to be modified. To create a damageable object, assign the entity script. This gives the object stats such as health, and the object will be disabled once the health reaches zero. To make any entity a functioning enemy, assign the pathfinding script and assign an item that can deal damage such as the pistol to the entity script. It will now patrol, chase and attack at set ranges according to variably defined parameters. This allows for new enemies to be created without requiring testing or the modification of code.

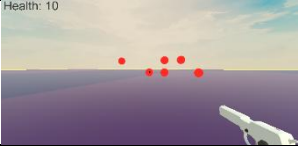
Finally, as for management of users and data, this has a planned structure to make maintenance and further modification simple. Every item and player in the database uses the same data structure, to allow them to be stored all together. The players and items are split into two tables and linked with another table. This allows new items to be added to the database by only adding one line to the items table. It can then be reused

as many times as needed on as many players as needed. The creation of new players has been automated through a php script which means no developer maintenance is required in this section. However, this script has not been integrated with the game yet so creation must be done manually for now.

In conclusion, maintenance in most of the 4 areas has been streamlined so that as little code must be modified as possible. This allows for minimal testing or accidental bugs, meaning maintenance should be very simple. However in the database, it has a structure ready for simple maintenance but needs some more work to be fully functional.

7.3 Success Criteria Recap

Success criteria requirement	Evidence	?
Different random level every time.		✓
The tiles join neatly.	When the central tile is a different size to the others, gaps are left in the generation. This does not happen when the central tile is the same size. This was overlooked in testing.	X
Adjustable level sizes.		✓
Adjustable rarity of unique tiles.		✓
Accounts can be created.	A php script to create player accounts was made, however I never got around to implementing it into the actual game. Therefore, accounts can't be created in-game.	X
Players can log in to accounts.	 This demonstrates a successful login where data for that user is pulled from the server and entered into the player class.	✓
Data about items can be retrieved.	The php script for this was created and functions but cannot be interacted with in game yet.	X
Player statistics are visible but non-modifiable.	Any scripts that modify player data require the player's password, correct, as an input to run successfully.	✓
The player has functional, user-friendly movement.	The player moves with WASD. The movement is snappy and responds to inputs without delay so seems natural and is intuitive.	✓
Items can be used by the player.	 The pistol item can be used and visualised with a 3D model, while the health consumable can also be used but is not visualised.	✓

Items loaded from the database.	This was not implemented.	X
Stationary targets can be created.	 These are stationary targets that can be destroyed when damaged.	✓
Enemies move autonomously.	Enemies select random points within their range of movement then move to that point.	✓
Enemies can chase the player.	If you move within a certain range of an enemy, they will spot and chase you.	✓
Enemies can use their held item.	When within range of the enemy they will use their equipped item. Pistols were equipped when checking this so the player was destroyed.	✓
Anything can be damageable by using a script.	Applying the entity script to any object allows that object to be damaged and destroyed, and to store data like stats.	✓

Total requirements met: 12/16

7.4 Summary

12 out of 16 requirements for the project were met which is a 75% success rate. The other components have either not been started or have had the framework laid out for them allowing them to be completed with relative simplicity in the future. This shows that the systems planned to be implemented were implemented and work as desired.

The project itself was designed to be a rogue like shooter with a database integration, and the project at this stage has set out a solid basis to build upon. All the main systems such as player movement, items and their interactions, enemies, and most of the server functionality have been implemented. This allows each of these components to be used and put together to create a full game experience that people can enjoy.

In conclusion, while not providing a full playable game yet, the project has laid out a framework for the game to be built upon, with minimal more code interaction required, simply using the created systems.

Bibliography

Section	Name	Source
1.41	Destiny 2	https://bungie.net/
1.4.1	DIM	https://destinyitemmanager.com/
1.4.2	Soul Knight	https://soul-knight.fandom.com/wiki/Soul_Knight_Wiki
1.6	C#	https://learn.microsoft.com/en-us/dotnet/csharp/
1.6	PHP	https://en.wikipedia.org/wiki/PHP
1.6	MySQL	https://en.wikipedia.org/wiki/MySQL
1.6	XAMPP	https://www.apachefriends.org/
1.6	Unity3D	https://unity.com/
1.6	Blender	https://www.blender.org/
1.6	VS Code	https://code.visualstudio.com/
1.6.2	UnityWebRequest	https://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.html
2.1.1	Rigidbody	https://docs.unity3d.com/ScriptReference/Rigidbody.html
5.3.2	Camera.main	https://docs.unity3d.com/ScriptReference/Camera-main.html
5.3.2	Layermask	https://docs.unity3d.com/ScriptReference/LayerMask.html
6.2.1	Physics.Raycast	https://docs.unity3d.com/ScriptReference/Physics.Raycast.html
6.2.2	V3.MoveTowards	https://docs.unity3d.com/ScriptReference/Vector3.MoveTowards.html