

MARCH 17, 2023

MULTIPLAYER CREATIVE SANDBOX

DRACOTT, PERCY
HILLS ROAD

Table of Contents

Analysis	3
Project Overview	3
Project Outline	3
My Client	3
An Interview With My Client	3
Objectives	4
Similar Games	4
Research	5
Outline of Solution	5
World Generation Prototype	6
Entity Relationship Diagram	8
System Flowchart	9
Unity Engine Framework	10
Required Runtime Data	10
Asset Requirements	12
Documented Design	12
Function Abstraction	12
User Interface Diagram	13
Algorithms	15
Map Generation	15
Adding Caves	16
Adding Trees	19
Map Saving and Loading	21
Player Control and Spawning	23
Zombie Control	26
Client and Server Scripts	28
Miscellaneous Scripts	34
Technical Solution	35
Server Scripts	35
NetworkManager.cs	35
MessageExtension.cs	38
GameLogic.cs	39
Player.cs	41
Zombie.cs	45

GenerationScriptV2.cs	47
WorldEventManager.cs	60
SpawnController.cs	61
ZombieControl.cs	62
DayNightCycle.cs	67
Client Scripts	69
NetworkManager.cs	69
MessageExtension.cs	71
GameLogic.cs	73
UIManager.cs	74
Player.cs	75
Zombie.cs	79
GenerationScriptV2.cs	80
PlayerController.cs	86
BlockInteractions.cs	88
HealthManager.cs	93
AttackManager.cs	95
UsernameText.cs	96
HudManager.cs	97
ItemInHandScript.cs	99
PlayerMenuManager.cs	99
PlayerTextChat.cs	101
TextChatManager.cs	102
WorldEventManager.cs	103
ZombieControl.cs	104
Assets	107
Technique Table	111
Testing	112
Server Testing Table	112
Server Test Evidence	113
Client Testing Table	116
Client Testing Evidence	118
Evaluation	124
Solution Overview	124
Objective Evaluation	124
Client Feedback	125
Potential Improvements	126
Source Tracker	127

Analysis

Project Overview

In this project, I will be exploring a Solution to a problem, to fulfil the needs of my client. Video game, since their creating have been a means of socialising, connecting and relaxing. A recent boom in popularity, owing to people spending more time gaming since the recent Covid-19 pandemic and lockdowns, has resulted in games becoming a staple in many people's lives. This ever-growing popularity creates a strong demand for new and unique games to enjoy.

Project Outline

My client is an avid video game enjoyer, who like many enjoy the social aspect of video games. They require a multiplayer game in the "Creative Sandbox" genre, so that they can play and create with their friends.

While this may be available through other games, their requirements for a game that runs well on very low powered hardware so that they can run the game on their laptops in a LAN configuration, vastly limits the choices of game. This excludes games such as terraria or Minecraft.

With all this in mind, the client has made it clear that they want a complicated and optimised map generation algorithm as well as multiplayer capability, as connecting on the same session is the most core component, and the most decorated feature.

My Client

My client in question is an avid video game fan. They particularly like retro style games, with simple and well executed features.

An Interview with Elliot - My Client

Q1. How would you like the game to be played?

A1: The Game needs to be playable on PC, as my friends and I all have laptops which we play our games on. Therefore, the controls need to be suited to a keyboard and mouse, and with a trackpad friendly control scheme.

Q2. How would you like the game to be used and by who?

A2: The game will be played by me and my friends using multiplayer. In either a LAN with all players on my Wi-Fi, or over a WAN hosted by me as I know how to set my router up for Port-Forwarding.

Q3. Which aspects of a "Creative Sandbox" do you find most enjoyable?

A3: For me I find the building aspect of the game most enjoyable. Coming up with an idea for a build. then creating it in 'reality' most appeals to me. I also enjoy exploring the game world, as I can often find inspiration of new builds in the landscape.

Q4. Which features would you most associated with a “Creative Sandbox”?

A4: In my opinion, gathering resources for a build through mining is a key feature of a creative game. As it rewards the player with a real sense of achievement and satisfaction, knowing the work it takes to complete their build. I also enjoy the sense of progression, as getting new tools makes once difficult tasks seem trivial.

Q5. Which areas of a “Creative Sandbox” map are you most interested in?

A5: Of course, the landscape of a map is important, as much of the gameplay occurs there. However, I find exploring the thrill of exploring cave systems in games to be the most enjoyable.

Objectives

01. Have functioning game element, which:
 - 1.1. Allows the player to move around the world,
 - 1.2. Contains hostile Mobile-Entities (mobs) in the world,
 - 1.3. Has some form of combat between the player and the mobs,
 - 1.4. Has a unique procedurally generated terrain and cave system,
 - 1.5. Has a destructible world where blocks can be mined by the player,
 - 1.6. Has a building system where the player can build structures,
02. Have a randomly generating map, which:
 - 2.1. Is randomly generated at game start,
 - 2.2. Has a tunnelling cave generation algorithm,
 - 2.3. Has randomly generated trees,
03. Have a functioning Menu System, which:
 - 3.1. Allows for new games to be created,
 - 3.2. Allows for players to connect online,
 - 3.3. Displays the game title,
 - 3.4. Allows the user to close the program,
04. Have a functioning Multiplayer System, which:
 - 4.1. Allows multiple players to play in the same world,
 - 4.2. Allows the world to be synced across each player,
 - 4.3. Allows interactions between players,
 - 4.4. Allow players to connect through LAN or WAN
05. Satisfy the clients brief by:
 - 5.1. Have a save-able and load-able world,
 - 5.2. Have a Day / Night cycle,
 - 5.3. Include fantasy mobs such as skeletons, or zombies,
 - 5.4. Be playable on PC,
 - 5.5. Be playable on a mouse and keyboard or trackpad,
06. Be appealing to my End-Users by:
 - 6.1. Having a consistent art style throughout,
 - 6.2. Using a consistent pixel by pixel tile size throughout,

Similar Games

In order to create my program and explore possible features, I have researched into 2 existing “Creative Sandbox” games. Minecraft and Terraria.

Minecraft	Terraria
-----------	----------

<ul style="list-style-type: none"> • Made for PC and other platforms.= • 3D • Able to create LAN multiplayer games, however there's no inbuilt way to play WAN multiplayer for free. • Uses 16x16 pixel art style. • Procedurally generated map with cave system. 	<ul style="list-style-type: none"> • Made of PC and other platforms • 2D • Can create free multiplayer games. • Uses pixel art textures. • Side Scroller • Items to enhance your character. •
--	--

Research

From these notes I have devised that my "Creative Sandbox" game will:

- Be a side-scrolling 2D game, to reduce computational load on a system.
- Use Keyboard and Mouse/Trackpad controls.
- Feature pixel art graphics (8p8px).
- Have free WAN multiplayer capability.
- Include a uniquely generating map and cave system.
- Include crafting/progression to enhance a player's character.
- Include game saving where the world can be saved and loaded from the server.

Additional Advanced Features

- Be able to save and load player inventories.

My solution will require two applications, a server and client. My client will run the server application and then, they can connect using client applications.

Outline of Solution

Random Map Generation

To meet the client's brief, I will be making use of random map generation, meaning that a uniquely generated map can be made for each new game. In order to do this, I will need to find a way to randomly generate a map.

Although, Minecraft is a 3D game its' early beta methods for world generation are great to analyse. As it is made for 3D, an altered 2D version of its' system producing a single slice would be highly efficient. For my research of Procedural generation therefore, I will have researched the solution used by beta versions of Minecraft. Minecraft uses a Perlin noise function to give the height at each given coordinate, in addition to many other functions to add trees and caves when creating a map. As my game will be 2D I will have to research other methods of generation of cave systems and other world features. For cave generation, I will produce a prototype to explore different generation methods; giving me a chance to analyse and evaluate each solution. In this I will explore both a Perlin noise approach and an iterative cellular automata approach.

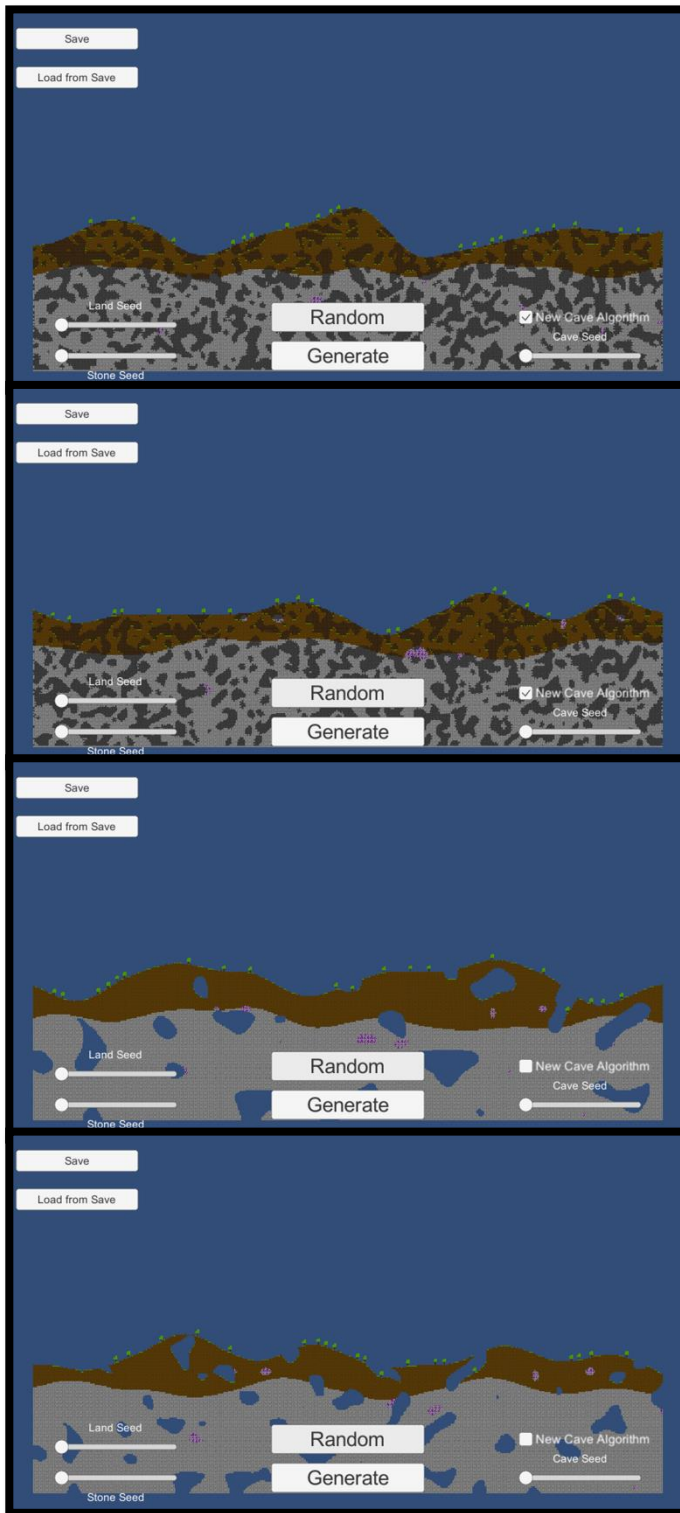
Multiplayer

For multiplayer I will be using Riptide Networking through Unity, I chose this as it offers a high degree of control of network entities through code, with open-source code. For my

client, I will produce two applications, a server and client build. With clients many clients connecting to one server. Clients will have to send the position of their player to the server as player movement will be managed client side. Ordinarily this is a bad idea for a multiplayer game, as this makes cheating much easier; a cheater can simply change the data in messages sent to the server, however Elliot has said that this won't be a problem as only a few trusted friends will have access to the game. As well as this, managing movement client side will spread the processing load across multiple systems in a thick-client type solution, reducing the load placed on the computer running the server application, Elliot notes that he is likely to run both the client and server applications simultaneously on his laptop and therefore optimisation of the networking side is key.

World Generation Prototype

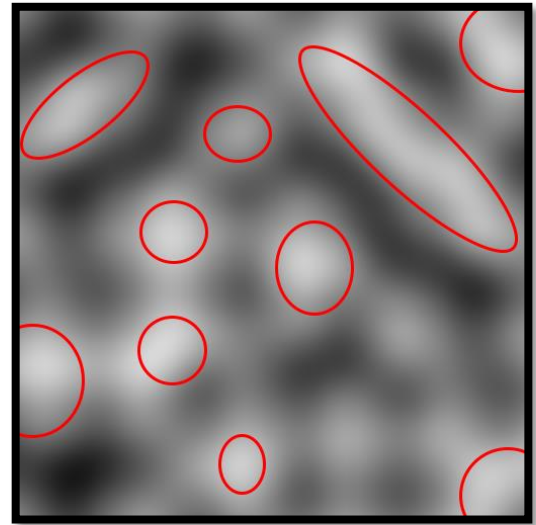
In this prototype, terrain generation is implemented using Perlin noise, by giving the function the x-coordinate and a seed value instead of a y-coordinate, for each value of x. This produces a smooth and varied terrain, before using Perlin noise I had experimented with using a Sin() based function to generate terrain, however this led to a far too periodic looking terrain with repeating hills and few areas of flat land. Elliot says it's important for



varied terrain to keep worlds looking interesting, however having a few flatter spaces is ideal for building.

Trees are added randomly, using a random integer value for the x-position, then checked for collisions to prevent trees from spawning onto of each other.

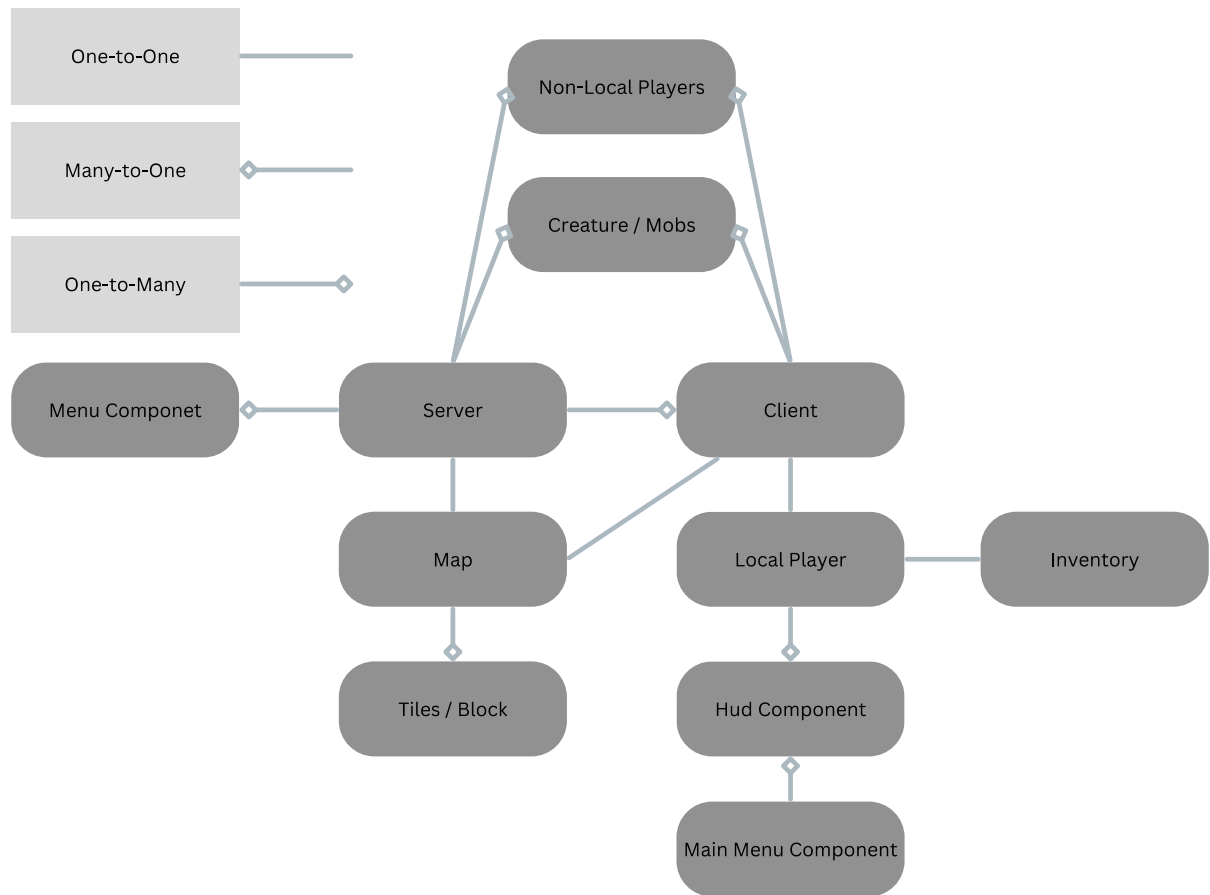
The first two screenshots show a cellular automata approach to cave generation. This code will be explained in Documented Design.



Screenshots 3 and 4 show the Perlin noise cave generation, where parts of the terrain are cut out using a Perlin noise map. Areas on the map are cut out if the resulting Perlin noise output of their position are above a threshold value, as shown. Any area of the map within the ellipse will be cut out as caves.

Elliot says that he greatly prefers the Cellular Automata based cave generation algorithm, as the caves produced by the algorithm are far more complex and varied.

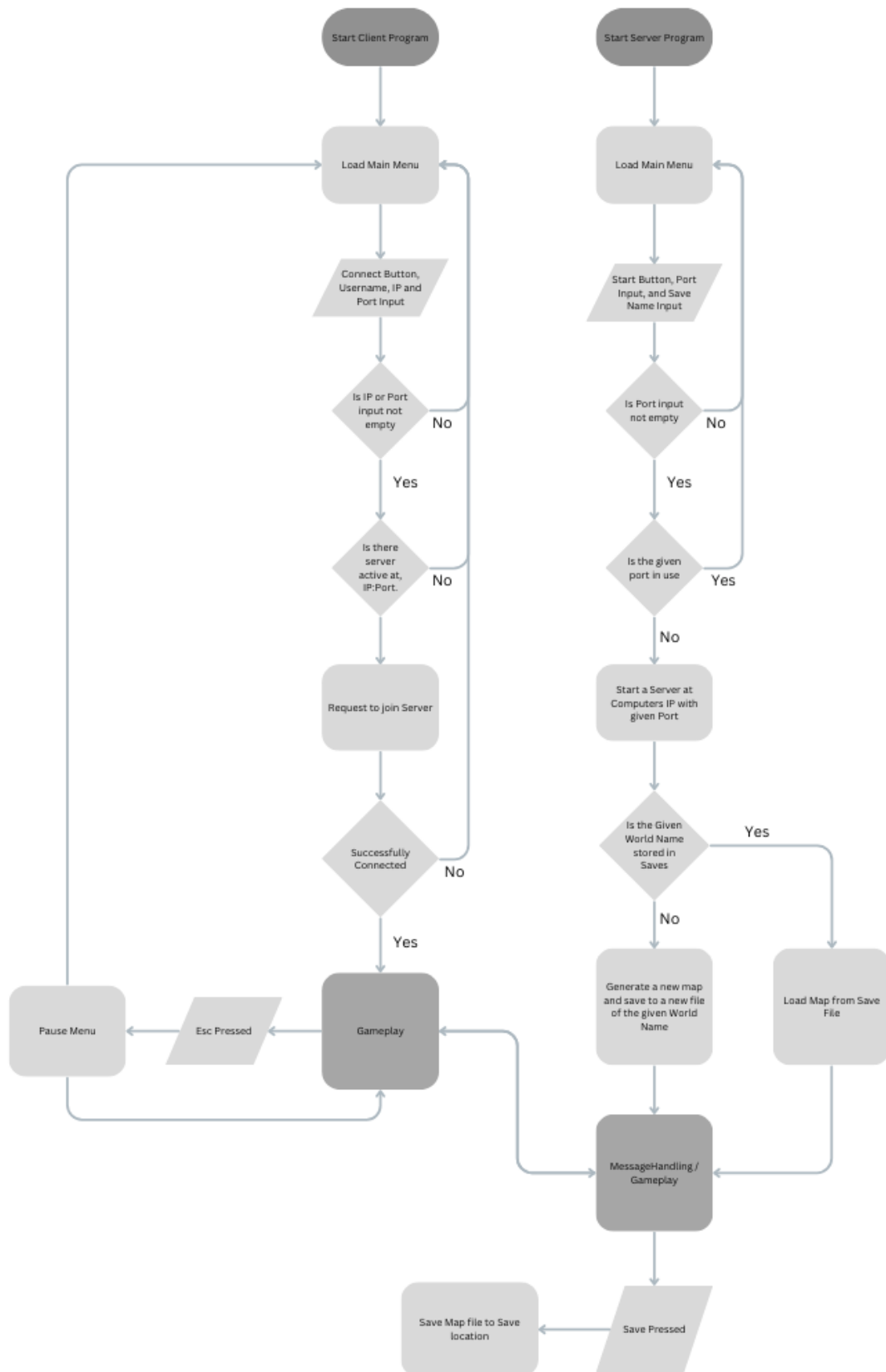
Entity Relationship & Scene Diagram



Within Unity entities, known as “game objects”, can be added to scenes, using the `Instantiate()` function. As shown above, on the client side a local player will be added to a scene in addition to non-local players. A local player contains all the control and interaction scripts which takes in the users’ inputs. A non-local player is a projection of another clients’ local player, whose position is sent to all other clients through the server to update the position of its’ corresponding non-local player.

Map, Mobs and Non-Local players, will be shared between both Server and all Clients connected. With. Mobs being controlled, spawned and respawned by the server.

System Flowchart



Unity Engine Framework

Within any unity script, derived from `UnityEngine.MonoBehaviour` there are predetermined methods such as:

- The Method `Start()`, that runs on an entities' first frame.
- The Method `Away()`, that runs before an entity is loaded into a scene.
- The Method `Update()`, that runs on every frame.
- The Method `FixedUpdate()`, that runs 60 times per second.

Most importantly within unity, individual scripts, and other components can be placed onto objects within a game scene. These include colliders, physics bodies (known as a `Rigidbody`) as well as textures.

Required Runtime Data

Player GameObject		
Data Name/ Access Name	Data Type	Represents
transform.position	Vector3	Position of the player in the scene
transform.scale	Vector3	The overall scale of the player in the scene
transform.localScale	Vector3	The scale of the player relative to its current scale
playerHealth	integer	Current player health
inventory	Integer array	An array of the current block held in the inventory
hasAxe	bool	Whether the player has a Pickaxe
hasSword	bool	Whether the player has a Sword
itemHolding	integer	The ordinal number for the inventory array
ScreenToWorldPoint(Input.mousePosition)	Vector3	Returns the position of the mouse in relation to its position in the physical scene
velocity	integer	The velocity of the player in the scene
OnGround	bool	A physics overlap at the players feet to determine if they are on the ground
Id	ushort	The players Id in the scene

IsLocal	bool	Whether the player is the Local player
---------	------	--

Zombie GameObject		
Data Name/ Access Name	Data Type	Represents
transform.position	Vector3	Position of the entity in the scene
transform.scale	Vector3	The overall scale of the entity in the scene
transform.localScale	Vector3	The scale of the entity relative to its current scale
playerInViewRange	bool	Whether a player is in sight range
PlayerInAttackRange	bool	Whether there's a player in the attacking range
closestPlayer	Vector3	The closest player entity to the zombie
Id	ushort	The Id of the zombie in the scene

Input field Component		
Data Name/ Access Name	Data Type	Represents
text	string	The text held in the interactable area
position	Vector2	Position on the UI canvas

Button Component		
Data Name/ Access Name	Data Type	Represents
DisplayText	string	The text held in the text area
position	Vector2	Position on the UI canvas
colour	Color	The colour applied to the texture of the button

Tile Component		
Data Name/ Access Name	Data Type	Represents
TileMap	TileMap	The tilemap the tile belongs to.
Tile	RuleTile	The current tile being rendered

position	Vector2	The position of the tile in the TileMap's grid
tilemapCollider	Collider	The physics collider of the tile

Hud Component		
Data Name/ Access Name	Data Type	Represents
text	string	The text held in the text area
position	Vector2	Position on the UI canvas
texture	Texture	Current rendered texture

Asset Requirements

Text fonts used will be acquired from www.dafont.com, specifically using fonts that are completely royalty free, all fonts used will be credited in the source tracker.

All textures are of my own making, made using Adobe Photoshop.

Design

Function Abstraction – From Flowchart

Server	
Function	Definition
Load Main Menu	All Menu Components, and background images in the scene are enabled, input text boxes are unlocked
Request to Start Server	Port Number is taken in from the input field and passed into the constructor for the Server class
Generate New Map	Landscape and Rock seeds are randomly generated. Empty 2D array of map size and width is generated. Perlin noise-based generation function and cellular automata cave generation function is run. This is copied to the 2d array and then rendered to the Tile map.
Load Map from Save	A new Stream Reader is created, then this cycles through each line of the save .txt file and the added to the 2D map array.
Save Map	A new Stream Writer is created. Foreach row in the 2D map array, the content of the row is written to the file with StreamWriter.Write().

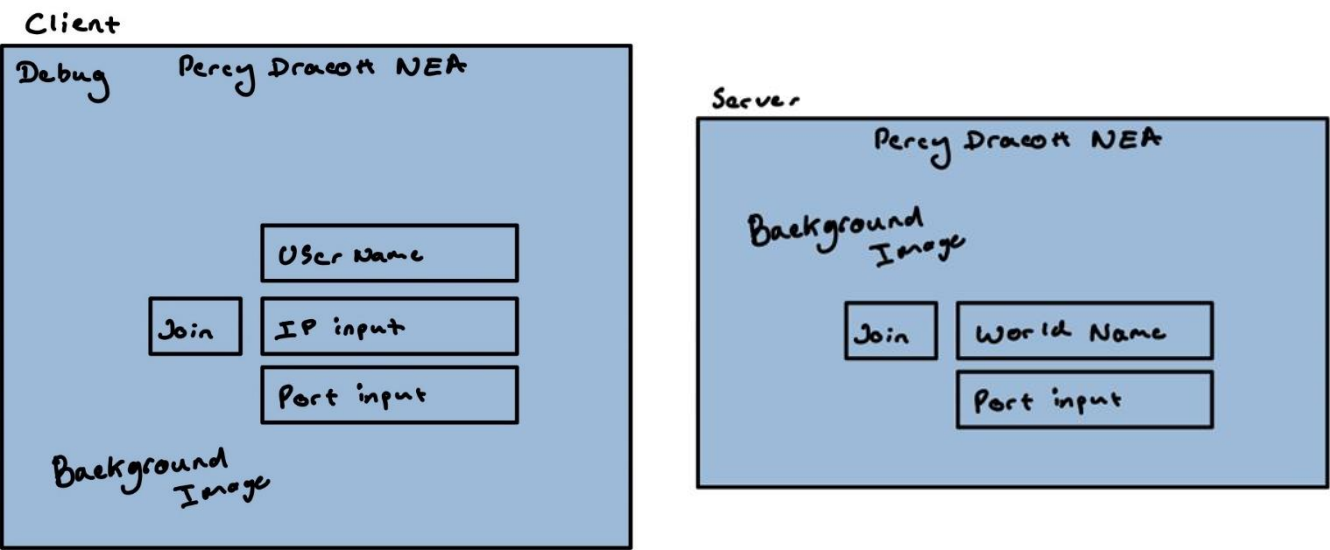
Gameplay	Many functions will run on gameplay, MessageHandling will include syncing player positions, block placing and breaking, in addition to passing the map file between the server and client upon joining.
----------	---

Client	
Function	Definition
Load Main Menu	All Menu Components, and background images in the scene are enabled, input text boxes are unlocked
Request to Join	Port number and IP address will be taken in as a ushort and string from the input fields. All menu Components are disabled, and input fields are locked. A request is sent to join any server at the given Ip and Port. This will return a bool; a failed connection will reload the Main Menu.
Gameplay	First the map will be loaded from the server and then a player spawned. Positions of the local player and any modifications made to the map will be sent to the server.
Pause Menu	Pressing the Esc key will toggle the pause menu. On a press, all pause menu components will be enabled and all player controlling scripts will be disabled. This will allow a player to resume or leave the server, returning them to the main menu.

User Interface Diagram

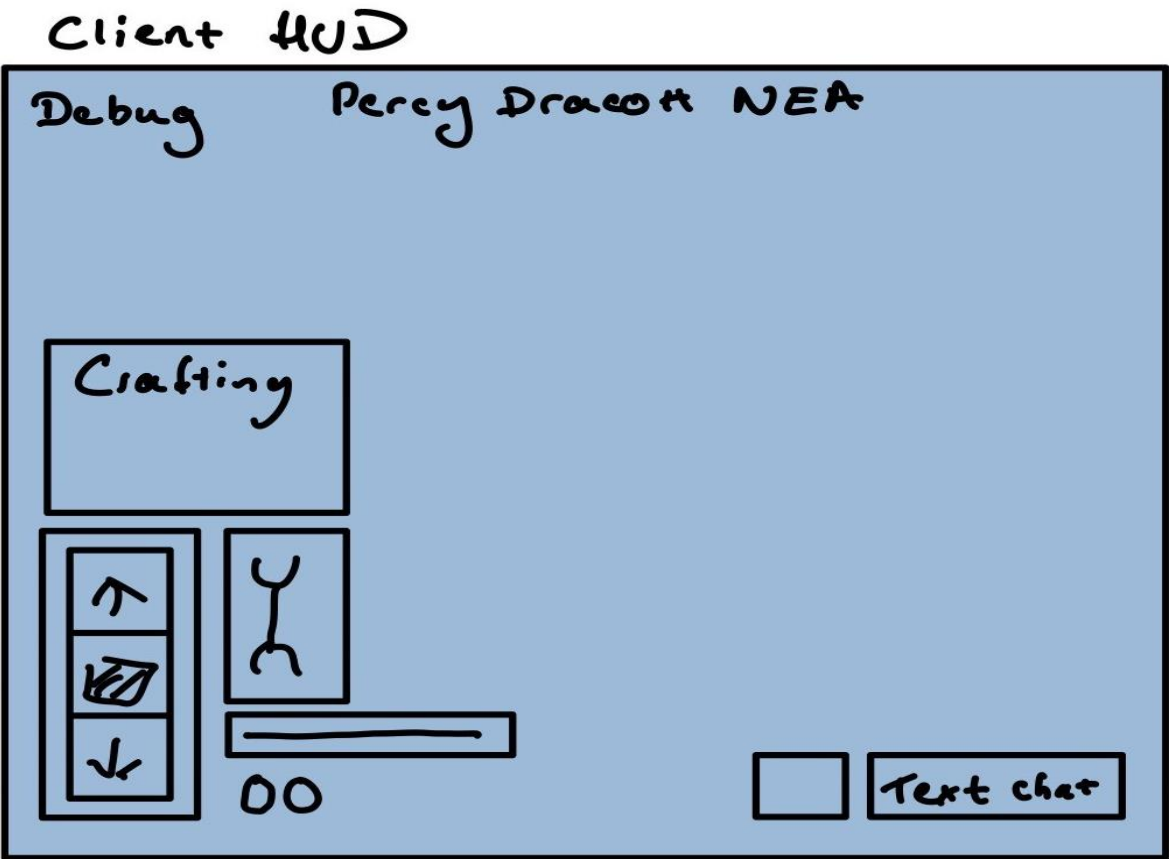
Throughout the design phase, I have sent various diagrams, algorithms, and interface designs to Elliot, my client, to see if he thinks they meet the objectives, or if they need to be improved on. I will have different versions of these throughout the document.

Here's the initial and final designs for the user interface; main menu and HUD.



The Main menu scene of both the client and server applications will obviously contain a few collections of graphics, including text components, buttons, and input fields. These will all need to be unloaded when a game is started.

The HUD layout, as planned is below,



The HUD layout contains a prototype for the inventory system, where the player can scroll through items represented by the shaded square. Each element on the HUD drawing will be rendered above the gameplay. Below the crafting icon, which will toggle the 'crafting' element, is the health bar which will display the hp of the local player.

Algorithms

I've given each algorithm a rough difficulty level to implement, **easy**, **medium**, or **hard** – the harder algorithms may need more revisions as other elements of the design, such as data structures or data storage, develop.

Map Generation

As I've made a world generation prototype, I will go over the procedures and functions contained in that first.

Name: Generation	Definition: Overall function that call each part of the generation script, and initialises the map	
Input: worldWidth, worldHeight		Output: N/A
<pre>Subroutine Generation(worldWidth, worldHeight) ForegroundTileMap.ClearAllTiles() BackgroundTileMap.ClearAllTiles() Seed = Random.WithinRange(0,1000) StoneSeed = Random.WithinRange(0,1000) CaveSeed = Random.WithinRange(0.02,0.05) OreSeed = Random.WithinRange(0.03,0.05) map = new int[worldWidth,worldHeight] If IsFromSave() LoadMap() SaveMap() else OptimisedTerrainGeneration(map, worldWidth, worldHeight, GrassSoil, Stone) ApplyCaves(map); AddTrees(TreePopulation, TestTileFG, Log, Leaf) Endif Renderer(map, TestTileFG, TestTileBG) GenerateSaveFile(worldName) SaveMap() endSubroutine</pre>		
Relevant Objective: 2.1		Time Complexity: O(1)
Difficulty: medium		

Name: TerrainGeneration	Definition: A function that creates the soil and stone layer of the map, in addition to adding ores. This is done in a loops per column of the map for efficiency.	
Input: worldWidth, worldHeight, worldMap, GrassTexture, StoneTexture	Output: N/A	
<pre>Subroutine TerrainGeneration(worldWidth, worldHeight, worldMap, Grass, Stone) perlinNoiseSoil = 0 perlinNoiseStone = 0 For x = 0, x < worldWidth, x++ perlinNoiseSoil = RoundToInt(PerlinNoise(x / (smoothness), Seed) * worldHeight / 5) perlinNoiseSoil += worldHeight / 3 For y = 0, y < perlinNoiseSoil, y++ worldMap[x,y] = 1 Endfor perlinNoiseStone = RoundToInt(PerlinNoise(x / (smoothness), StoneSeed) * worldHeight / 8) perlinNoiseStone += worldHeight / 4 For y = 0, y < perlinNoiseStone, y++ worldMap[x,y] = 2 Endfor For y = 0, y < perlinNoiseSoil, y++ PerlinNoiseOre = RoundToInt(PerlinNoise(x * OreSeed, y * OreSeed)) If PerlinNoiseOre <= OreThreshhold map[x,y] = 3 Endif Endfor Endfor endSubroutine</pre>		
Relevant Objective: 2.1	Time Complexity: $O(n^2)$	
Difficulty: medium		

Adding Caves

Adding Caves was specified by my clients as a major aspect of the generation system, therefore Elliot and I had, many conversations to find the most appropriate generation system that fitted with his expectations.

Name: ApplyCaves Version 1	Definition: Function that cuts caves out of the 2D array map file.	
Input: worldWidth, worldHeight, worldMap	Output: N/A	
Subroutine ApplyCaves(worldWidth, worldHeight, worldMap)		

<pre> For x = 0, x < worldWidth, x++ For y = 0, y < worldHeight, y++ PerlinNoiseCave = RoundToInt(PerlinNoise(x * CaveSeed, y * CaveSeed)) If PerlinNoiseCave >= CaveThreshold map[x,y] = 0 Endif Endfor Endfor endSubroutine </pre>	
Relevant Objective: 2.2	Time Complexity: $O(n^2)$
Difficulty: <i>easy</i>	

Name: ApplyCaves Version 2	Definition: Function that cuts caves out of the 2D array map file, using cellular automata	
Input: worldWidth, worldHeight, worldMap	Output: N/A	
<pre>Subroutine ApplyCaves(worldWidth, worldHeight, worldMap) RandomFillGeneration() Automata(iterations) For x = 0, x < worldWidth, x++ For y = 0, y < worldHeight, y++ int topOfWorld = RoundToInt(PerlinNoise(x / (smoothness), Seed) * worldHeight / 5) perlinNoiseSoil += worldHeight / 3 If cavemap[x, y] == 0 && y != 0 && y != topofworld - 1 && y != topofworld - 2 If map[x,y] == 1 map[x,y] = 8 Endif If map[x,y] == 2 OR map[x,y] == 3 map[x,y] = 9 Endif Endif Endfor Endfor endSubroutine</pre>		
Relevant Objective: 2.2	Time Complexity: $O(n^2)$	
Difficulty: easy		

Name: RandomFillGeneration	Definition: Function that fills a 2D array randomly with a random fill percentage.	
Input: worldWidth, worldHeight	Output: N/A	

Subroutine RandomFillGeneration(worldWidth, worldHeight, caveMap) caveMap = new int[worldHeight, worldWidth] For x = 0, x < worldWidth, x++ For y = 0, y < worldHeight, y++ caveMap[x,y] = Random.WithinRange(0,100) < RandomFillPercentage ? 1 : 0 Endfor Endfor endSubroutine	
Relevant Objective: 2.2	Time Complexity: $O(n^2)$
Difficulty: easy	

Name: Automata	Definition: Function that fills a 2D array randomly with a random fill percentage.	
Input: iterationCount, worldWidth, worldHeight	Output: N/A	
<pre>Subroutine Automata(iterations, worldWidth, worldHeight) For i = 0, i < count, i++ int[,] tempMap = caveMap.Clone For xs = 0, xs < worldWidth - 1, xs++ For ys = 0, ys < worldHeight - 1, ys++ int neighbours = 0 neighbours = GetNeighbours(tempMap, xs, ys, neighbours) If neighbours > 4 caveMap[xs,ys] = 1 else caveMap[xs,ys] = 0 Endif Endfor Endfor Endfor endSubroutine</pre>		
Relevant Objective: 2.2	Time Complexity: $O(n^2)$	
Difficulty: hard		

Name: GetNeighbours	Definition: Finds the number of neighbouring active cells in the array, around a given point.	
Input: tempMap, xs, ys, neighbours		Output: int neighbours
Subroutine GetNeighbours(tempMap, xs, ys, neighbours) For x = xs - 1, x <= xs, x++ For y = ys - 1, y <= ys, y++		

<pre> If y <= worldHeight AND x <= worldWidth If y != ys AND x != xs If tempMap[x,y] = 1 neighbours++ Endif Endif Endif Endfor Endfor Return neighbours endSubroutine</pre>	
Relevant Objective: 2.2	Time Complexity: O(1)
Difficulty: medium	

Adding Trees

Adding Trees was also pointed out by Elliot for part of the world generation, in addition this opens avenues for weapon crafting and addition building blocks that can be obtained through crafting, such as the plank block.

Elliot notes that have blocks of multiple colours and textures is important for keeping builds looking interesting.

This algorithm ensures that trees are not placed atop each other, by checking the position of each past tree. This is done by looping through the positions of all placed trees and subtracting the x-coordinate of a proposed tree, from the position of a placed tree, and ensuring that the absolute value of the sum is greater than 6; as this is the padding value given so trees aren't overly clustered.

Name: AddTrees	Definition: Adds a given number of trees to the surface of the terrain, without placing then atop one another
Input: worldWidth, worldHeight, DensityCount	Output: N/A
<pre>Subroutine AddTrees(worldWidth, worldHeight, Map, density) //Check for overpopulation If density >= worldWidth / 6 density = worldWidth / 6 Endif int temporaryX = 0 int[] positionHistory = new int[density] For i = 0, x < density, i++ bool collisionAvoidance = true While(collisionAvoidance) int holdingPositionVariable = Random.WithinRange(2, worldWidth - 2) collisionAvoidance = false</pre>	

<pre>For j = 0, j <= i, j++ If AbsoluteValue(holdingPositionVariable - positionHistory[j]) < 6 collisionAvoidance = true Endif Endfor If collisionAvoidance == False temporaryX = holdingPositionVariable Endif Endwhile positionHistory[i] = temporaryX int temporaryY = worldHeight - 1 While Map[temporaryX,temporaryY - 1] == 0 temporaryY-- Endwhile Map[temporaryX,temporaryY] = 4 Map[temporaryX, temporaryY + 1] = 4 Map[temporaryX, temporaryY + 2] = 5 Map[temporaryX, temporaryY + 3] = 5 Map[temporaryX, temporaryY + 4] = 5 Map[temporaryX + 1, temporaryY + 2] = 5 Map[temporaryX - 1, temporaryY + 2] = 5 Map[temporaryX + 1, temporaryY + 3] = 5 Map[temporaryX - 1, temporaryY + 3] = 5 Map[temporaryX + 1, temporaryY + 4] = 5 Endfor endSubroutine</pre>	
Relevant Objective: 2.3	Time Complexity: O(n!)
Difficulty: hard	

Name: Renderer	Definition: Function that takes the 2D map array and displays the content to the Tile map	
Input: worldWidth, worldHeight, worldMap		Output: N/A
<pre>Subroutine Renderer(worldMap) ForegroundTileMap.ClearAllTiles() BackgroundTileMap.ClearAllTiles() For x = 0, x < worldMap.GetHorizontalLength, x++ For y = 0, y < worldMap.GetVerticalLength, y++ If worldMap[x,y] == 1 ForegroundTileMap.SetTile(new Vector3Int(x,y,0), GrassSoil) BackgroundTileMap.SetTile(new Vector3Int(x,y,0), GrassSoilBackground) Endif If worldMap[x,y] == 2</pre>		

<pre> ForegroundTileMap.SetTile(new Vector3Int(x,y,0), Stone) BackgroundTileMap.SetTile(new Vector3Int(x,y,0), StoneBackground) Endif If worldMap[x,y] == 3 ForegroundTileMap.SetTile(new Vector3Int(x,y,0), Ore) BackgroundTileMap.SetTile(new Vector3Int(x,y,0), StoneBackground) Endif If worldMap[x,y] == 4 ForegroundTileMap.SetTile(new Vector3Int(x,y,0), Log) Endif If worldMap[x,y] == 5 ForegroundTileMap.SetTile(new Vector3Int(x,y,0), Leaf) Endif If worldMap[x,y] == 6 ForegroundTileMap.SetTile(new Vector3Int(x,y,0), Plank) Endif If worldMap[x,y] == 8 BackgroundTileMap.SetTile(new Vector3Int(x,y,0), GrassSoilBackground) Endif If worldMap[x,y] == 9 BackgroundTileMap.SetTile(new Vector3Int(x,y,0), StoneBackground) Endif Endfor Endfor endSubroutine</pre>	
Relevant Objective: 2.1	Time Complexity: $O(n^2)$
Difficulty: easy	

Map Saving and Loading

Elliot notes that the ability to save and load past maps is important, as it will enable him and his friends to spend multiple sessions on one project.

Name: SaveMap	Definition: Function that saves the 2D map array to a file	
Input: worldMap		Output: N/A
<pre>Subroutine SaveMap(worldMap) Using StreamWriter sw = new StreamWriter(\$"WorldSaves/{worldName}/worldSave.txt") For y = 0, y < worldMap.GetVerticalLength, y++ For x = 0, x < worldMap.GetHorizontalLength, x++ sw.Write(worldMap[x,y]) Endfor sw.WriteLine() Endfor</pre>		

EndUsing endSubroutine	
Relevant Objective: 5.1	Time Complexity: $O(n^2)$
Difficulty: medium	

The map will be located in a save folder: WorldSaves/{worldName}/worldSave.txt, where worldName is a string variable passed in from the input fields upon starting a server.

Name: LoadMap	Definition: Function that fills an empty 2D array with the map data.
Input: worldMap	Output: N/A
Subroutine LoadMap(worldMap) int y = 0 Using StreamReader sr = new StreamReader(\$"WorldSaves/{worldName}/worldSave.txt") string value While (value = sr.ReadLine()) != Null For x = 0, x < worldMap.GetHorizontalLength, x++ worldMap[x,y] = ConvertToByte(value[x]) - 48 Endfor y++ Endwhile EndUsing Renderer(worldMap) endSubroutine	
Relevant Objective: 5.1	Time Complexity: $O(n^2)$
Difficulty: medium	

The following two functions will be used on the server application to check if a map is being loaded from a save or is new, and then create the saving directory if necessary.

Name: GenerateSaveFile	Definition: Function that creates the file saving directory upon loading a new world, not a previous save.
Input: worldName	Output: N/A
Subroutine GenerateSaveFile(worldName) if File.Exists(\$"WorldSaves/{worldName}") == false Directory.Create(\$"WorldSaves/{worldName}") Endif endSubroutine	
Relevant Objective: 5.1	Time Complexity: $O(1)$
Difficulty: easy	

Name: IsFromSave	Definition: Function that returns a Boolean if the given world name has a file directory	
Input: worldName		Output: Boolean
<pre>Subroutine Bool IsFromSave(worldName) if File.Exists(\$"WorldSaves/{worldName}") return true else return false Endif endSubroutine</pre>		
Relevant Objective: 5.1		Time Complexity: O(1)
Difficulty: medium		

Player Control and Spawning

Local players will need to be controlled by the player, interact with the world, and have a health system.

Each of the following function will be part of PlayerController.cs, a script that will be run on the local player in a scene.

Name: xMovePlayer	Definition: Function that uses the keyboard input though unity and sets the velocity of the player. This function is called in the Update() Function of unity.	
Input: N/A	Output: n/A	
<pre>Subroutine xMovePlayer() float moveX = Input.GetInputsFromAxis("Horizontal") Vector2 movement = new Vector2(moveX,0) if PlayerIsOnGround() Player.Velocity = movement * speedValue Elself PlayerIsOnGround() == false AND moveX != 0 Player.Velocity = moveX * speedValue / 2 Endif endSubroutine</pre>		
Relevant Objective: 1.1	Time Complexity: O(1)	
Difficulty: easy		

Name: PlayerBreakingBlock	Definition: Function that uses the mouse position in the world to remove a block at its given position, with an amount of time based on the type of block. A call for this function will be run in Update(), if the mouse button is depressed.
---------------------------	--

Input: timeMining, mousePos	Output: n/A
<pre> Subroutine PlayerBreakBlock(timeMining, mousePos) if timeMining == 0 Vector2 mousePosOnCall = new Vector2(truncate(mousePos.x), truncate(mousePos.y)) Endif if mousePosOnCall = new Vector2(truncate(mousePos.x), truncate(mousePos.y)) timeMining += Time.DeltaTime //Counts the time since a frame where Time.DeltaTime wasn't called. if timeMining > AmountOfTimeToBreak() BreakBlock() timeMining = 0 Endif Else timeMining = 0 Endif endSubroutine </pre>	
Relevant Objective: 1.5	Time Complexity: O(1)
Difficulty: medium	

Name: AmountOfTimeToBreak	Definition: Function that takes the mouse position, to find the type of block in the 2D map array at the position. Returning the amount of time, it takes to break the block, at the requested position.
Input: mousePos	Output: float
<pre> Subroutine float AmountOfTimeToBreak(mousePos) float timetobreak = 0 Switch (Map.ReturnBlockType(new Vector2(truncate(mousePos.x), truncate(mousePos.y)))) Case 1 timetobreak = 0.6f break Case 2 timetobreak = 1.8f break Case 3 timetobreak = 1.8f break Case 4 timetobreak = 1f break Case 5 timetobreak = 0.2f break Case 6 </pre>	

<pre> timetobreak = 1f break Default timetobreak = 1f break EndSwitch if player.HasAxe return timetobreak / 2f Else return timetobreak Endif endSubroutine</pre>	
Relevant Objective: 1.5	Time Complexity: O(1)
Difficulty: easy	

The additional features functions, of player inventory loading and saving are planned out below.

Name: SavePlayerState	Definition: Function that save the players inventory, items, and position to a file.
Input: worldName, PlayerID	Output: n/A
<pre>Subroutine SavePlayerState(worldName, PlayerID) Using StreamWriter sw = new StreamWriter(\$"WorldSaves/{worldName}/{PlayerID}.txt") For i = 0, i < inventory.Length, i++ sw.WriteLine(inventory[i]) Endfor sw.WriteLine(transform.positon.x) sw.WriteLine(transform.positon.y) if Player.HasAxe sw.WriteLine("1") Else sw.WriteLine("0") Endif if Player.HasSword sw.WriteLine("1") Else sw.WriteLine("0") Endif EndUsing endSubroutine</pre>	
Relevant Objective: 1.5	Time Complexity: O(1)
Difficulty: medium	

Name: LoadPlayerState	Definition: Function that takes a save file and fills the players inventory, items, and position.
Input: worldName, PlayerID	Output: n/A
<pre> Subroutine LoadPlayerState(worldName, PlayerID) string value int i = 0 float tempx float tempy Using StreamReader sr = new StreamReader(\$"WorldSaves/{worldName}/{PlayerID}.txt") While (value = sr.ReadLine()) != null if i < 10 inventory[i] = Convert.ToInt16(value) Endif if i == 10 tempx = (float)Convert.ToDouble(value) Endif if i == 11 tempy = (float)Convert.ToDouble(value) Endif if i == 12 AND value == "1" Player.HasAxe = True Else Player.HasAxe = False Endif if i == 13 AND value == "1" Player.HasSword = True Else Player.HasSword = False Endif i++ Endwhile transform.positon = new Vector2(tempx, tempy) EndUsing endSubroutine </pre>	
Relevant Objective: 1.5	Time Complexity: O(1)
Difficulty: medium	

Zombie Control

Zombie movement will be handled by the server, and positions will be passed to the clients. Zombie attacks, however, will be calculated client side as the zombie will attack when a player is within range, as guided by the server's movement.

Name: IdlePatrol	Definition: Function that directs a zombie on a patrolling walk when a player is not in range
Input: playerInRange, patrolRange	Output: n/A
<pre> Subroutine IdlePatrol(playerInRange, patrolRange) float lengthOfTime = walkSpeed * patrolRange if (patrolRange > 0) AND !playerInRange timeSinceActive += Time.deltaTime; if timeSinceActive > 2 * lengthOfTime timeSinceActive -= 2 * lengthOfTime Endif if timeSinceActive > lengthOfTime zombieRB.velocity = new Vector2(walkSpeed * transform.localScale.x, zombieRB.velocity.y) Else zombieRB.velocity = new Vector2(-walkSpeed * transform.localScale.x, zombieRB.velocity.y) Endif Endif timeSinceActive = 0 endSubroutine </pre>	
Relevant Objective: 1.5	Time Complexity: O(1)
Difficulty: medium	

Name: PlayerInRangePositon	Definition: Function that returns the position of the closest player to the zombie, within sight range.
Input: playerInRange, patrolRange	Output: Vector2
<pre> Subroutine Vector2 playerInRangePosition() Vector2 closePlayer = new Vector2(0,0) float previousDistance = sightRange If playerInRange Colliders[] players = (Physics2D.OverlapCircleAll(transform.position, sightRange, playerLayer) float currentDistance Foreach var item in players currentDistance = item.Magnitude //The overall distance of the Vector2, when treated as a coordinate Vector2 If currentDistance < previousDistance closePlayer = item.transform.position previousDistance = currentDistance Endif EndForeach Endif Return closePlayer endSubroutine </pre>	

Relevant Objective: 1.5	Time Complexity: O(1)
Difficulty: medium	

Client and Server Scripts

For implementing multiplayer, I will be using Riptide Networking [1], a lightweight C# networking library primarily designed for use in multiplayer games. This enables me to start servers and connect clients using the Server and Client classes, in addition sending messages between Server and Client.

I will be using the Riptides message system and message handling to sync parts of the applications, such as player positions, map files, and map changes. As I am new to Riptide Networking, I will be following a basic tutorial [2] to set up a simple client and server, and to learn the basics.

In Riptide, messages have a capacity of 1200 bytes, in addition to send modes reliable and unreliable. Reliable will check that the data has been received using majority voting and is therefore more taxing on the network as 3 transmissions are sent for each message; I will therefore only be using Reliable send mode for passing the map and spawning / despawning objects.

A message is also sent with an Enum, in the header to tell the receiver which MessageHandler to use upon receiving the message. A message can be sent to a specific client as shown below with NetworkManager.Instance.Server.Send(message, toClientId), or to all with SendToAll, however this can only be done by the server.

As the map is 1024 blocks wide, the map file must be a byte array to save space in the message payload. This is done as the 2D array is sent in rows.

I will also be adding an Override/ Extension to the Message class to allow the transmissions of Vector2 and Vector3 for ease of use.

Name: PassingMapToClient	Definition: Function that is called on a new client connecting to pass the 2D map array to the client.
Input:	Output:
Server Side	
<pre>Subroutine SendMap(ushort toClientId) byte[,] mapToSend = MapManager.SendMap() For int y, y < mapToSend.GetLength(1), y++ Message message = Message.Create(MessageSendMode.reliable, (ushort)ServerToClientId.map) byte[] tempXS = new byte[mapToSend.GetLength(0)] For int x, mapToSend.GetLength(0), x++ tempXS[x] = mapToSend[x, y] Endfor message.AddShort(y) message.AddInt(FindObjectOfType<GenerationScriptV2>().worldWidth) message.AddBytes(tempXS, false, true) NetworkManager.Instance.Server.Send(message, toClientId)</pre>	

Endfor endSubroutine	
Client Side	
<pre>[MessageHandler((ushort)ServerToClientId.map)] Subroutine GettingMapFromServer(Message message) short Layer = message.GetShort() int lengthofByteArray = message.GetInt() byte[] ByteArray = new byte[lengthofByteArray] message.GetBytes(lengthofByteArray, ByteArray) CallMapBuild(ByteArray, Layer) message.Release() endSubroutine //Within the Generation Script Subroutine CallMapBuild(byte[] mapSlice, short layer) For int x = 0, x < map.GetLength(0), x++ map[x,layer] = mapSlice[x] Endfor If layer = worldHeight - 1 Renderer(map) Endif endSubroutine</pre>	
Relevant Objective: 4.2	Time Complexity: $O(n^2)$
Difficulty: hard	

Syncing of Non-Local players first starts with each clients sending the position of their Local player to the server, which stores each of their positions as Non-Local players within the server monitoring scene. It can then send the positions of these out to all clients along with the Id's of each, the clients can then check to see if the Id matches their own and if not move the Non-Local players around. Messages will be sent in Unreliable send mode as losing a position will not matter, as a new and more up to date position will be sent before a resend is possible.

Name: SyncNonLocalPlayer	Definition: Function that is called on in FixedUpdate(), to send the position all players within the server scene to the clients.
Input: N/A	Output: N/A
Server Side	
<pre>Subroutine SyncNonLocalPlayers() Foreach Player player in list.Values Message message = Message.Create(MessageSendMode.unreliable, (ushort)ServerToClientId.syncNonLocalPosition) message.AddUShort(players.Id); message.AddVector3(players.gameObject.transform.position) NetworkManager.Instance.Server.SendToAll(message) Endforeach</pre>	

endSubroutine	
Client Side	
Static Subroutine SyncingPlayers(Message message) ushort playerId = message.GetUShort() Vector3 playerPosition = message.GetVector3() If playerId != NetworkManager.Instance.Client.Id Player.list[playerID].gameObject.transform.position = playerPosition Endif message.Release() endSubroutine	
Relevant Objective: 4.1	Time Complexity: O(n)
Difficulty: hard	

Players both Local and Non-Local will be stored in a Dictionary, with the value being their Player.cs class and the key being their network Id. This is done as the local player can be found easily by comparing the dictionary key to the NetworkManager.Instance.Client.Id

Name: SpawnPlayer	Definition: Function that takes is triggered by a message from the server to spawn a new player onto the Client and add them to their dictionary.
Input: ushort id, string username, Vector3 position	Output: N/A
Client Side	
Static Subroutine Spawn(ushort id, string username, Vector3 position) Player player If id = NetworkManager.Instance.Client.Id player = Instantiate(GameLogic.Instance.LocalPlayer, position, Quaternion.identity).GetComponent<Player>() player.IsLocal = true player.SetUserNameText = username Else player = Instantiate(GameLogic.Instance.Non-LocalPlayer, position, Quaternion.identity).GetComponent<Player>() player.IsLocal = false player.SetUserNameText = username Endif player.name = \$"Player {id} ({(string.IsNullOrEmpty(username) ? "Guest" : username)})" player.Id = id player.userName = username list.Add(id, player) endSubroutine	
Relevant Objective: 4.1	Time Complexity: O(1)
Difficulty: hard	

Here's all the Enum's I expect to need to use within the networking code:

Name: SpawnPlayer	Definition: Enums that can be added to the head of a message, to determine which message handler the message is sent to.	
Input: N/A		Output: N/A
<pre>public Enum ServerToClientId : ushort playerSpawned = 1, map, syncNonLocalPosition, syncMapUpdate, lightPosition, zombieSpawning, zombiePosition, zombieDeath, textChat, endEnum public Enum ClientToServerId : ushort name = 1, updatePlayerPosition, updateServerMap, zombieDeath, updateTextChat, endEnum</pre>		
Relevant Objective: 4.1		Time Complexity: N/A
Difficulty: medium		

First the `SendBlockUpdateToServer(int x, int y, byte block)`, function is called by a client on interacting with the map, this then sends a message to the Server to update its map. The server then sends a message to all clients other than the origin of the initial message to update their maps.

Name: SyncMaps	Definition: Function that is called on a client building or breaking parts of the map, to sync the changes across all clients.	
Input: int x, int y, byte block		Output: N/A
Client Side		
<pre>Subroutine SendBlockUpdateToServer(int x, int y, byte block) RiptideNetworking.Message message = Message.Create(MessageSendMode.reliable, (ushort)ClientToServerId.updateServerMap) message.AddInt(x) message.AddInt(y) message.AddByte(block) NetworkManager.Instance.Client.Send(message) //Runs UpdateMap() on Server Side endSubroutine</pre>		


```
[MessageHandler((ushort)(ServerToClientId.syncMapUpdate))]
```

```
Static Subroutine SyncingMaps(Message message)
```

```
    int xPos = message.GetInt()
```

```
    int yPos = message.GetInt()
```

```
    byte block = message.GetByte()
```

```
    MapManager.ServerUpdatingBlock(block, xPos, yPos)
```

```
    message.Release()
```

```
endSubroutine
```

Server Side

```
[MessageHandler((ushort)ClientToServerId.updateServerMap)]
```

```
Static Subroutine UpdateMap(ushort fromClientId, Message message)
```

```
    int xPos = message.GetInt()
```

```
    int yPos = message.GetInt()
```

```
    byte block = message.GetByte()
```

```
    MapManager.ServerUpdatingBlock(block, xPos, yPos)
```

```
    message.Release()
```

```
    UpdatePlayerMaps(block, xPos, yPos, fromClientId) //Runs UpdatePlayerMaps()
```

```
endSubroutine
```

```
Static Subroutine UpdatePlayerMaps(byte block, int xPos, int yPos, ushort originPlayer)
```

```
    Foreach (Player players in list.Values)
```

```
        if (players.Id != originPlayer)
```

```
            RiptideNetworking.Message message =
```

```
Message.Create(MessageSendMode.reliable, (ushort)ServerToClientId.syncMapUpdate)
```

```
    message.AddInt(xPos)
```

```
    message.AddInt(yPos)
```

```
    message.AddByte(block)
```

```
    NetworkManager.Instance.Server.Send(message, players.Id) //Runs
```

```
SyncingMaps() on Client Side
```

```
    Endif
```

```
    Endforeach
```

```
endSubroutine
```

Relevant Objective: 4.1	Time Complexity: O(n)
Difficulty: hard	

As Zombies need to be synced across all clients, including spawning, movement and despawning; they will need to be created and handled similarly to Non-Local player within a client.

In addition, zombies will need to be spawned into a client upon connecting, if for example they join during the night. Zombies will also be stored in a dictionary with the value being the Zombie class and the Key being an Id given on zombie spawn.

As shown before in **Zombie Control**, the movement of the zombies is controlled on the server and send to each client from within the Zombie class within Zombie.cs.

This Function is similar to the one used to sync player positions with to the server, however the client sends the position, and this is taken in by the server and sent to all other clients to update the Non-Local player in their scene.

Name: SendPositon	Definition: Function that runs in FixedUpdate() to send the position of a zombie to all clients.
Input: N/A	Output: N/A
Server Side	
Subroutine SendPositionToClients() Message message = Message.Create(MessageSendMode.unreliable, (usshort)ServerToClientId.zombiePosition) message.AddUShort(Id) message.AddVector3(list[Id].gameObject.transform.position) NetworkManager.Instance.Server.SendToAll(message) endSubroutine	
Client Side	
Subroutine SendPositionToClients() Message message = Message.Create(MessageSendMode.unreliable, (usshort)ServerToClientId.zombiePosition) message.AddUShort(Id) message.AddVector3(list[Id].gameObject.transform.position) NetworkManager.Instance.Server.SendToAll(message) endSubroutine	
Relevant Objective: 5.3	Time Complexity: O(1)
Difficulty: medium	

Zombie spawning will be handled by the server, as shown below:

Name: SpawnZombies	Definition: Function that runs in FixedUpdate(), and spawns zombies on the change of day to night.
Input: N/A	Output: N/A
Server Side	
Subroutine SpawnZombies() If !DayNightCycleInUse.isDay() AND GenerationScriptInUse.terrainGenerationComplete AND !hasSpawnedMobs ushort zombieId = 0 Foreach var position in FindSpawnLocations(spawnAmount) FindObjectOfType<GameLogic>().CallZombieSpawn(positionElement, zombieId) zombieId++ Endforeach hasSpawnedMobs = true Endif If DayNightCycleInUse.isDay hasSpawnedMobs = false Endif	

endSubroutine	
Relevant Objective: 5.3	Time Complexity: O(1)
Difficulty: medium	

Name: FindSpawnLocations	Definition: Function that finds a given number of spawn locations
Input: spawnAmount	Output: Vector3[]
Server Side	
Subroutine FindSpawnLocations(spawnAmount) Vector3[] spawnLocations = new Vector3[spawnAmount] For i = 0, i < spawnAmount, i++ int tempPos = Random.WithinRange(1, GenerationScriptInUse.worldWidth) spawnLocations[i] = new Vector3(tempPos., GenerationScriptInUse.ReturnGroundPosition(tempPos), 0) Endfor Return spawnLocations endSubroutine	
Relevant Objective: 5.3	Time Complexity: O(n)
Difficulty: medium	

Miscellaneous Scripts

This includes some other gameplay scripts such as the text chat. Text Chat will be implemented using a Queue of string as this will be the easiest way to get the chat to stack in the correct order within the text component.

Name: AddToChat	Definition: Function that takes a text input and adds it to a queue, with overflow protection.
Input: message	Output: N/A
Client Side	
Subroutine AddToChat(string Message) If messages.Count == chatLength messages.Dequeue() messages.Enqueue(Message) Else messages.Enqueue(Message) Endif endSubroutine	
Relevant Objective: 6.0	Time Complexity: O(1)
Difficulty: easy	

Name: UpdateChat	Definition: Function that updates the content of the text element.
Input: N/A	Output: N/A

Client Side	
<pre>Subroutine updateChat() textBox.text = "" int count = 1 Foreach var item in messages textBox.text += "\n" + count.ToString() + " : " + item count++ Endforeach endSubroutine</pre>	
Relevant Objective: 6.0	Time Complexity: O(1)
Difficulty: easy	

Technical Solution

All code and Assets, including the project file can be found on my GitHub: [\[x\]](#)

Server Scripts

Runs On GameObject: NetworkManager
Name: NetworkManager.cs
<pre>using RiptideNetworking; using RiptideNetworking.Utls; using System; using System.Collections.Generic; using TMPro; //using UnityEditor.VersionControl; using UnityEngine; using UnityEngine.UI; //using static UnityEditor.Progress; using Message = RiptideNetworking.Message; /// <summary> /// Enumerators used in message headers to direct it to the correct Message Handler /// </summary> public enum ServerToClientId : ushort { playerSpawned = 1, map, syncNonLocalPosition, syncMapUpdate, lightPosition, zombieSpawning, zombiePosition, zombieDeath, textChat, despawnPlayer, } public enum ClientToServerId : ushort {</pre>

```

name = 1,
updatePlayerPosition,
updateServerMap,
zombieDeath,
updateTextChat,
}

public class NetworkManager : MonoBehaviour
{
    private static NetworkManager instance;
    /// <summary>
    /// A Singleton spawned in on the first action of the scene, a singleton being a class that
    allows only a single instance of itself to be created and gives access to that created instance.
    /// </summary>
    public static NetworkManager Instance
    {
        get => instance;
        private set
        {
            if (instance == null) instance = value;
            else if (instance != value)
            {
                Debug.Log($"{nameof(NetworkManager)} instance already exists, destroying new");
                Destroy(value);
            }
        }
    }

    /// <summary>
    /// Serialised fields show up in the unity editor window without being public variables, and
    are therefore protected.
    /// </summary>
    public Server Server { get; private set; }
    [SerializeField] private ushort port;
    [SerializeField] private ushort maxClientCount;

    //[SerializeField] private TMP_InputField ipField;
    [SerializeField] private TMP_InputField portField;
    [SerializeField] private Button CurrentServerState;

    /// <summary>
    /// Runs before being loaded.
    /// </summary>
    private void Awake()
    {
        instance = this;
        Application.runInBackground = true;
    }

    /// <summary>
    /// Runs on its first frame of existence, Methods are added to the Servers.ClientDisconnected
    function so they are also run on the event
    /// </summary>
    private void Start()
    {
        //Application.targetFrameRate = 60;
        RiptideLogger.Initialize(Debug.Log, Debug.Log, Debug.LogWarning, Debug.LogError, false);
        Server = new Server();
        //Server.Start(port, maxClientCount);
    }
}

```

```

        Server.ClientDisconnected += PlayerLeft;
        //Server.ClientConnected += SendMap;
    }

    /// <summary>
    /// Runs 60 times per second.
    /// </summary>
    private void FixedUpdate()
    {
        Server.Tick();
        if (Server.IsRunning) CurrentServerState.GetComponent<Image>().color = Color.green;
        else CurrentServerState.GetComponent<Image>().color = Color.red;
    }

    private void OnApplicationQuit()
    {
        Server.Stop();
    }

    /// <summary>
    /// Removes the player from the scene if they leave.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void PlayerLeft(object sender, ClientDisconnectedEventArgs e)
    {
        Message message = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.despawnPlayer);
        message.AddUShort(e.Id);
        NetworkManager.Instance.Server.SendToAll(message);
        Destroy(Player.list[e.Id].gameObject);
    }

    /// <summary>
    /// Used to start a server from the UI within the scene
    /// </summary>
    public void StartFromButtons()
    {
        if (!string.IsNullOrEmpty(portField.text))
        {
            port = Convert.ToUInt16(portField.text);
            //port = Convert.ToUInt16(ipField.text);
            Server.Start(port, maxClientCount);
        }
    }

    /// <summary>
    /// Used as a function accessible to the stop button.
    /// </summary>
    public void StopFromButton()
    {
        Server.Stop();
        foreach (var item in Zombie.list.Values)
        {
            Destroy(item.gameObject);
        }
    }

```

```

        FindObjectOfType<GenerationScriptV2>().ClearMap();
        FindObjectOfType<DayNightCycle>().gameObject.transform.position = new Vector3(512, 4000,
0);
    }

    //[MessageHandler((ushort)ClientId.updatePlayerPosition)]
    //private static void PlayerPos(Message message)
    //{
    //    Debug.Log("Received Call");
    //}
}

```

Runs On GameObject: N/A //From to RiptideNetworking GitHub

Name: MessageExtension.cs

```

using RiptideNetworking;
using UnityEngine;

public static class MessageExtensions
{
    #region Vector2
    /// <inheritdoc cref="Add(Message, Vector2)"/>
    /// <remarks>Relying on the correct Add overload being chosen based on the parameter type can
increase the odds of accidental type mismatches when retrieving data from a message. This method
calls <see cref="Add(Message, Vector2)"/> and simply provides an alternative type-explicit way to
add a <see cref="Vector2"/> to the message.</remarks>
    public static Message AddVector2(this Message message, Vector2 value) => Add(message, value);

    /// <summary>Adds a <see cref="Vector2"/> to the message.</summary>
    /// <param name="value">The <see cref="Vector2"/> to add.</param>
    /// <returns>The message that the <see cref="Vector2"/> was added to.</returns>
    public static Message Add(this Message message, Vector2 value)
    {
        message.AddFloat(value.x);
        message.AddFloat(value.y);
        return message;
    }

    /// <summary>Retrieves a <see cref="Vector2"/> from the message.</summary>
    /// <returns>The <see cref="Vector2"/> that was retrieved.</returns>
    public static Vector2 GetVector2(this Message message)
    {
        return new Vector2(message.GetFloat(), message.GetFloat());
    }
    #endregion

    #region Vector3
    /// <inheritdoc cref="Add(Message, Vector3)"/>
    /// <remarks>Relying on the correct Add overload being chosen based on the parameter type can
increase the odds of accidental type mismatches when retrieving data from a message. This method

```

```

calls <see cref="Add(Message, Vector3)"/> and simply provides an alternative type-explicit way to
add a <see cref="Vector3"/> to the message.</remarks>
    public static Message AddVector3(this Message message, Vector3 value) => Add(message, value);

    /// <summary>Adds a <see cref="Vector3"/> to the message.</summary>
    /// <param name="value">The <see cref="Vector3"/> to add.</param>
    /// <returns>The message that the <see cref="Vector3"/> was added to.</returns>
    public static Message Add(this Message message, Vector3 value)
    {
        message.AddFloat(value.x);
        message.AddFloat(value.y);
        message.AddFloat(value.z);
        return message;
    }

    /// <summary>Retrieves a <see cref="Vector3"/> from the message.</summary>
    /// <returns>The <see cref="Vector3"/> that was retrieved.</returns>
    public static Vector3 GetVector3(this Message message)
    {
        return new Vector3(message.GetFloat(), message.GetFloat(), message.GetFloat());
    }
#endregion

#region Quaternion
    /// <inheritdoc cref="Add(Message, Quaternion)"/>
    /// <remarks>Relying on the correct Add overload being chosen based on the parameter type can
    increase the odds of accidental type mismatches when retrieving data from a message. This method
    calls <see cref="Add(Message, Quaternion)"/> and simply provides an alternative type-explicit way
    to add a <see cref="Quaternion"/> to the message.</remarks>
    public static Message AddQuaternion(this Message message, Quaternion value) => Add(message,
value);

    /// <summary>Adds a <see cref="Quaternion"/> to the message.</summary>
    /// <param name="value">The <see cref="Quaternion"/> to add.</param>
    /// <returns>The message that the <see cref="Quaternion"/> was added to.</returns>
    public static Message Add(this Message message, Quaternion value)
    {
        message.AddFloat(value.x);
        message.AddFloat(value.y);
        message.AddFloat(value.z);
        message.AddFloat(value.w);
        return message;
    }

    /// <summary>Retrieves a <see cref="Quaternion"/> from the message.</summary>
    /// <returns>The <see cref="Quaternion"/> that was retrieved.</returns>
    public static Quaternion GetQuaternion(this Message message)
    {
        return new Quaternion(message.GetFloat(), message.GetFloat(), message.GetFloat(),
message.GetFloat());
    }
#endregion
}

```

Runs On GameObject: NetworkManager

Name: GameLogic.cs


```

using RiptideNetworking;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameLogic : MonoBehaviour
{
    private static GameLogic instance;
    public bool gameIsRunning { get; set; }

    /// <summary>
    /// Singleton
    /// </summary>
    public static GameLogic Instance
    {
        get => instance;
        private set
        {
            if (instance == null) instance = value;
            else if (instance != value)
            {
                Debug.Log($"{nameof(GameLogic)} instance already exists, destroying new");
                Destroy(value);
            }
        }
    }

    /// <summary>
    /// Player Game object made indirectly accessible for security
    /// </summary>
    public GameObject PlayerPrefab => playerPrefab;

    [Header("Prefabs")]
    [SerializeField] private GameObject playerPrefab;
    [SerializeField] private GameObject zombiePrefab;

    /// <summary>
    /// Runs before being loaded
    /// </summary>
    private void Awake()
    {
        Instance = this;
    }

    float timeSinceActive = 0f;

    /// <summary>
    /// Runs 60 times per second
    /// </summary>
    private void FixedUpdate()
    {
        timeSinceActive += Time.deltaTime;
        if (timeSinceActive > 1.0f && FindObjectOfType<NetworkManager>().Server.IsRunning)
        {
            RiptideNetworking.Message message = Message.Create(MessageSendMode.unreliable,
(ushort)ServerToClientId.lightPosition);
            message.AddVector3(FindObjectOfType<DayNightCycle>().ReturnLightPosition());
            NetworkManager.Instance.Server.SendToAll(message);
            //Debug.Log("light Pos sent");
            timeSinceActive = 0f;
        }
    }
}

```

```

    }

    //Zombie Management

    /// <summary>
    /// Used to spawn zombies, is called from the zombie spawning script
    /// </summary>
    /// <param name="position"></param>
    /// <param name="Id"></param>
    public void CallZombieSpawn(Vector3 position, ushort Id)
    {
        Zombie zombie = Instantiate(zombiePrefab, position,
Quaternion.identity).GetComponent<Zombie>();
        zombie.SendId(Id);

        Message message = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.zombieSpawning);
        message.AddVector3(position);
        message.AddUShort(Id);
        NetworkManager.Instance.Server.SendToAll(message);

    }

}

```

Runs On GameObject: Player

Name: Player.cs

```

using RiptideNetworking;
using System.Collections.Generic;
using UnityEngine;
//using static UnityEditor.Progress;
//using static UnityEditor.Experimental.GraphView.GraphView;

public class Player : MonoBehaviour
{
    public static Dictionary<ushort, Player> list = new Dictionary<ushort, Player>();

    public ushort Id { get; private set; }
    public string userName { get; private set; }

    /// <summary>
    /// Removes the player from the dictionary if they are destroyed within the scene
    /// </summary>
    public void OnDestroy()
    {
        list.Remove(Id);
    }

    /// <summary>
    /// Called From within a message, by a Client attempting to join the server. Adds the player
    to the Dictionary, along with defining its' identifiers. This is also sent to all other players
    connected to spawn a Non-Local player
    /// </summary>
    /// <param name="id"></param>
    /// <param name="username"></param>
    public static void Spawn(ushort id, string username)
    {

```

```

        foreach (Player otherPlayers in list.Values)
        {
            otherPlayers.SendSpawn(id);
        }

        Player player = Instantiate(GameLogic.Instance.PlayerPrefab,
FindObjectOfType<GenerationScriptV2>().PlayerSpawnPoint(),
Quaternion.identity).GetComponent<Player>();
        player.name = $"Player {id} ({(string.IsNullOrEmpty(username) ? "Guest" : username)})";
        player.Id = id;
        player.userName = string.IsNullOrEmpty(username) ? "Guest" : username;

        player.SendMap(player.Id);
        player.SendMobs(player.Id);
        player.SendSpawn();
        list.Add(id, player);
    }

    /// <summary>
    /// SendSpawn and the Override if a variable is passed in are used to send the spawn location
of a new player to the new client or to all existing clients.
    /// </summary>
    private void SendSpawn()
    {
        RiptideNetworking.Message message = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.playerSpawned);

        NetworkManager.Instance.Server.SendToAll(AddSpawnData(message));
    }

    private void SendSpawn(ushort toClientId)
    {
        RiptideNetworking.Message message = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.playerSpawned);

        NetworkManager.Instance.Server.Send(AddSpawnData(message), toClientId);
    }

    /// <summary>
    /// This adds the Id, username and a Vector3 position for the new player to be spawned at.
    /// </summary>
    /// <param name="message"></param>
    /// <returns></returns>
    private Message AddSpawnData(Message message)
    {
        message.AddUShort(Id);
        message.AddString(userName);
        message.AddVector3(FindObjectOfType<GenerationScriptV2>().PlayerSpawnPoint());
        return message;
    }

    /// <summary>
    /// Used to send the spawning position of new mobs in a scene, in addition to tellin the
client to spawn mobs.
    /// </summary>
    /// <param name="toClientId"></param>
    private void SendMobs(ushort toClientId)
    {
        foreach (var item in Zombie.List.Values)
        {

```

```

        Message message = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.zombieSpawning);
        message.AddVector3(item.gameObject.transform.position);
        message.AddUShort(item.Id);
        NetworkManager.Instance.Server.Send(message, toClientId);
    }
}

/// <summary>
/// Sends the Map file from the Server to the client, layer by layer due to the capacity limit
of messages.
/// </summary>
/// <param name="toClientId"></param>
private void SendMap(ushort toClientId)
{
    byte[,] maptoSend = FindObjectOfType<GenerationScriptV2>().SendMap();

    //List<int> TempList = new List<int>();
    //int[] mapAs1DArray;
    //for (int y = 0; y < maptoSend.GetLength(0); y++)
    //{
    //    for (int x = 0; x < maptoSend.GetLength(1); x++)
    //    {
    //        TempList.Add(maptoSend[y, x]);
    //    }
    //}
    ////int[] temporaryarray = new int[maptoSend.GetLength(0)];
    ////temporaryarray[0] = maptoSend[i];
    ////mapAs1DArray = TempList.ToArray();

    //message.AddInt(maptoSend.GetLength(0));
    //message.AddInt(maptoSend.GetLength(1));
    ////message.AddInts(mapAs1DArray, false, true);
    ///
    for (short y = 0; y < maptoSend.GetLength(1); y++)
    {
        //message = Message.Create(MessageSendMode.reliable, (ushort)ServerToClientId.map);
        Message message = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.map);
        byte[] tempXS = new byte[maptoSend.GetLength(0)];
        for (int x = 0; x < maptoSend.GetLength(0); x++)
        {
            tempXS[x] = maptoSend[x, y];
        }
        //
        //Debug.LogAssertion(y);
        message.AddShort(y);
        message.AddInt(FindObjectOfType<GenerationScriptV2>().worldWidth);
        message.AddBytes(tempXS, false, true);
        //Debug.Log("sending map to client");
        NetworkManager.Instance.Server.Send(message, toClientId);
    }

    //Debug.Log("sending map to client");
    //NetworkManager.Instance.Server.Send(message, toClientId);

}

/// <summary>
/// Runs 60 times per second.

```

```

/// </summary>
private void FixedUpdate()
{
    SyncNonLocalPlayers();
}

/// <summary>
/// Sends the position of all players connected along with their respective Id's to all
players.
/// </summary>
private void SyncNonLocalPlayers()
{
    foreach (Player players in list.Values)
    {
        Message message = Message.Create(MessageSendMode.unreliable,
(ushort)ServerToClientId.syncNonLocalPosition);
        message.AddUShort(players.Id);
        message.AddVector3(players.gameObject.transform.position);
        NetworkManager.Instance.Server.SendToAll(message);
    }
}

/// <summary>
/// Sends updates to the map to all but the original editor of the map
/// </summary>
/// <param name="block"></param>
/// <param name="xPos"></param>
/// <param name="yPos"></param>
/// <param name="originPlayer"></param>
private static void UpdatePlayerMaps(byte block, int xPos, int yPos, ushort originPlayer)
{
    foreach (Player players in list.Values)
    {
        if (players.Id != originPlayer)
        {
            RiptideNetworking.Message message = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.syncMapUpdate);
            message.AddInt(xPos);
            message.AddInt(yPos);
            message.AddByte(block);
            NetworkManager.Instance.Server.Send(message, players.Id);
        }
    }
}

/// <summary>
/// Message handler handles changing the position of a clients Local player on the Server.
/// </summary>
/// <param name="fromClientId"></param>
/// <param name="message"></param>
[MessageHandler((ushort)(ClientToServerId.updatePlayerPosition))]
private static void PlayerPos(ushort fromClientId, Message message)
{
    Vector3 positionFromMessage = message.GetVector3();
    //ushort playerIdFromMessage = message.GetUShort();
    //Debug.Log($"CALLED POSITION UPDATE, {positionFromMessage.x}:{positionFromMessage.y}");
    list[fromClientId].gameObject.transform.position = positionFromMessage;
    message.Release();
}

/// <summary>

```

```

/// Sent from the client upon connecting
/// </summary>
/// <param name="fromClientId"></param>
/// <param name="message"></param>
[MessageHandler((ushort)(ClientToServerId.name))]
private static void Name(ushort fromClientId, Message message)
{
    Spawn(fromClientId, message.GetString());
}

/// <summary>
/// Updates the Servers map from a message from a Client.
/// </summary>
/// <param name="fromClientId"></param>
/// <param name="message"></param>
[MessageHandler((ushort)ClientToServerId.updateServerMap)]
private static void UpdateMap(ushort fromClientId, Message message)
{
    //Vector2 BlockPos = message.GetVector2();
    int xPos = message.GetInt();
    int yPos = message.GetInt();
    byte block = message.GetByte();
    //Debug.Log($"Block Update Called at {(int)BlockPos.x}:{(int)BlockPos.y} for {block}");
    FindObjectOfType<GenerationScriptV2>().ServerUpdatingBlock(block, xPos, yPos);
    message.Release();
    UpdatePlayerMaps(block, xPos, yPos, fromClientId);
}

/// <summary>
/// Updates the Text Chat of all clients but the origin of the chat.
/// </summary>
/// <param name="fromClientId"></param>
/// <param name="message"></param>
[MessageHandler((ushort)ClientToServerId.updateTextChat)]
private static void SendUpdatesToPlayers(ushort fromClientId, Message message)
{
    string text = message.GetString();

    foreach (var item in list.Values)
    {
        if (item.Id != fromClientId)
        {
            Message message2 = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.textChat);
            message2.AddUShort(item.Id);
            message2.AddString(text);

            NetworkManager.Instance.Server.Send(message2, item.Id);
        }
    }
}
}

```

Runs On GameObject: Zombie

Name: Zombie.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using RiptideNetworking;
using RiptideNetworking.Utls;
//using static UnityEditor.Progress;

public class Zombie : MonoBehaviour
{
    public static Dictionary<ushort, Zombie> list = new Dictionary<ushort, Zombie>();

    public ushort Id { get; private set; }

    private void OnDestroy()
    {
        list.Remove(Id);
    }

    /// <summary>
    /// Used to define and add each zombie to the Dictionary
    /// </summary>
    /// <param name="id"></param>
    public void SendId(ushort id)
    {
        Zombie zombie = GetComponent<Zombie>();
        zombie.Id = id;
        list.Add(id, zombie);
    }

    /// <summary>
    /// Runs 60 times per second
    /// </summary>
    private void FixedUpdate()
    {
        SendPositionToClients();
    }

    /// <summary>
    /// Sends the position of the zombie to all clients
    /// </summary>
    private void SendPositionToClients()
    {
        Message message = Message.Create(MessageSendMode.unreliable,
(ushort)ServerToClientId.zombiePosition);
        message.AddUShort(Id);
        message.AddVector3(list[Id].gameObject.transform.position);

        NetworkManager.Instance.Server.SendToAll(message);
    }

    /// <summary>
    /// Function despawns zombie and tells every client also do so
    /// </summary>
    public void Despawn()
    {
        Message message = Message.Create(MessageSendMode.reliable,
(ushort)ServerToClientId.zombieDeath);
        message.AddUShort(Id);
        NetworkManager.Instance.Server.SendToAll(message);
        Destroy(list[Id].gameObject);
    }
}

```

```

    /// <summary>
    /// Handles a zombie death and sends a new message tell all clients to despawn the killed
    zombie
    /// </summary>
    /// <param name="fromClientId"></param>
    /// <param name="message"></param>
    [MessageHandler((ushort)ClientId.zombieDeath)]
    private static void ZombieKilledByPlayer(ushort fromClientId, Message message)
    {
        ushort idOfDeath = message.GetUShort();

        Message message2 = Message.Create(MessageSendMode.reliable,
        (ushort)ServerToClientId.zombieDeath);
        message2.AddUShort(idOfDeath);
        NetworkManager.Instance.Server.SendToAll(message2);
        Destroy(list[idOfDeath].gameObject);
    }
}

```

Runs On GameObject: MapManager

Name: GenerationV2.cs

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Text;
using UnityEngine;
using UnityEngine.Tilemaps;
using UnityEngine.UI;

public class GenerationScriptV2 : MonoBehaviour
{
    // Start is called before the first frame update
    public int worldWidth = 1024;
    public int worldHeight = 256;
    public bool terrainGenerationComplete { get; private set; }

    [SerializeField] float Seed;
    [SerializeField] float StoneSeed;
    //[SerializeField] float CaveSeed;
    [SerializeField] float OreSeed;

    float Smoothness = 40;
    float OreThreshold = 0.1f;
    //float CaveThreshold = 0.7f;

    [SerializeField] TileBase GrassSoil;
    [SerializeField] TileBase Stone;
    [SerializeField] TileBase SoilBG;

```



```

[SerializeField] TileBase StoneBG;
[SerializeField] TileBase Ore;
[SerializeField] TileBase Log;
[SerializeField] TileBase Leaf;
[SerializeField] TileBase Plank;

public Tilemap TestTileFG;
public Tilemap TestTileBG;

[SerializeField] public Slider SeedSlider;
[SerializeField] public Slider StoneSlider;
//[SerializeField] public Slider CaveSlider;

[SerializeField] float RandomPercentFill;
[SerializeField] int iterations;
[SerializeField] int TreePopulation;

string worldName;

byte[,] map;
byte[,] cavemap;

void Start()
{
    worldWidth = 1024;
    worldHeight = 256;

    //terrainGenerationComplete = Generation();
}
/*
//void Update()
//{
//    if (Input.GetKeyDown(KeyCode.Space))
//    {
//        Debug.Log("Reloading");
//        Generation();
//    }

//public void SetWorldName(string name) { worldName = name; }

/// <summary>
/// Used
/// </summary>
/// <param name="x"></param>
/// <param name="y"></param>
/// <returns></returns>
///
public int BreakBlock(int x, int y)
{
    //Debug.Log($"breakblock called, position {x}, {y}");

    if (!(map[x, y] == 8 || map[x, y] == 9 || map[x, y] == 0))
    {
        TestTileFG.SetTile(new Vector3Int(x, y, 0), null);
        TestTileFG.RefreshTile(new Vector3Int(x, y, 0));
        int block = map[x,y];
        if (map[x, y] == 1)
        {
            map[x, y] = 8;
        }
    }
}

```

```

        else if (map[x, y] == 2 || map[x, y] == 3)
        {
            map[x, y] = 9;
        }
        else map[x, y] = 0;
        FindObjectOfType<AudioManager>().Play("Block Break");
        return block;
    }
    return 0;
}

public int ReturnTypeInfoOfBlock(int x, int y)
{
    if (!(map[x, y] == 8 || map[x, y] == 9 || map[x, y] == 0))
    {
        int block = map[x, y];
        return block;
    }
    return 0;
}

public bool BuildBlock(int block, int x, int y)
{
    //Debug.Log($"buildblock called, position {x}, {y}");
    if ((map[x-1,y] != 0 || map[x + 1, y] != 0 || map[x, y - 1] != 0 || map[x, y + 1] != 0) &&
    (map[x,y] == 0 || map[x, y] == 8 || map[x, y] == 9))
    {
        map[x, y] = (byte)block;
        TileBase placing = Plank;
        switch (block)
        {
            case 1:
                placing = GrassSoil;
                break;
            case 2:
                placing = Stone;
                break;
            case 4:
                placing = Log;
                break;
            case 5:
                placing = Leaf;
                break;
            case 6:
                placing = Plank;
                break;
            default:
                return false;
        }

        TestTileFG.SetTile(new Vector3Int(x, y, 0), placing);
        TestTileFG.RefreshTile(new Vector3Int(x, y, 0));
        FindObjectOfType<AudioManager>().Play("Block Place");
        return true;
    }
    return false;
}
*/

/// <summary>
/// Used the Player.cs script to update blocks on the map
/// </summary>

```

```

/// <param name="block"></param>
/// <param name="x"></param>
/// <param name="y"></param>
public void ServerUpdatingBlock(int block, int x, int y)
{
    map[x, y] = (byte)block;
    TileBase placing = Plank;
    switch (block)
    {
        case 0:
            placing = null;
            break;
        case 1:
            placing = GrassSoil;
            break;
        case 2:
            placing = Stone;
            break;
        case 4:
            placing = Log;
            break;
        case 5:
            placing = Leaf;
            break;
        case 6:
            placing = Plank;
            break;
        case 8:
            placing = SoilBG;
            break;
        case 9:
            placing = StoneBG;
            break;
        default:
            break;
    }

    Debug.Log($"At Position {x}:{y}, Map is now {map[x,y]}");
    if (block == 8 || block == 9)
    {
        TestTileBG.SetTile(new Vector3Int(x, y, 0), placing);
        TestTileFG.SetTile(new Vector3Int(x, y, 0), null);
        TestTileBG.RefreshTile(new Vector3Int(x, y, 0));
        TestTileFG.RefreshTile(new Vector3Int(x, y, 0));
    }
    else
    {
        TestTileFG.SetTile(new Vector3Int(x, y, 0), placing);
        TestTileFG.RefreshTile(new Vector3Int(x, y, 0));
        TestTileBG.RefreshTile(new Vector3Int(x, y, 0));
    }
    //Debug.Log(TestTileFG.GetTile(new Vector3Int(x, y, 0)).name);
}

//public void BuildBlock(int block, int x, int y)
//{
//    map[x, y] = block;

//    TestTileFG.SetTile(new Vector3Int(x, y, 0), null);
//}

public string CurrentWorldName() { return (worldName); }

```

```

//public void GenerateFromSlider()
//{
//    Seed = SeedSlider.value;
//    StoneSeed = StoneSlider.value;
//    CaveSeed = CaveSlider.value;

//    map = new byte[worldWidth, worldHeight];
//    OptimisedTerrainGeneration(map, worldWidth, worldHeight, GrassSoil, Stone);
//    ApplyCaves(map);
//    AddTrees(TreePopulation, TestTileFG, Log, Leaf);
//    Renderer(map, TestTileFG, TestTileBG);
//}

/// <summary>
/// Called to generate a new map or to load a map from a save based on the save name given.
/// </summary>
/// <param name="saveName"></param>
/// <returns></returns>
public bool Generation(string saveName)
{
    worldName = saveName;
    TestTileBG.ClearAllTiles();
    TestTileFG.ClearAllTiles();
    Seed = (float)UnityEngine.Random.Range(0.0f, 1000.0f);
    StoneSeed = (float)UnityEngine.Random.Range(0.0f, 1000.0f);
    //CaveSeed = (float)UnityEngine.Random.Range(0.02f, 0.05f);
    OreSeed = (float)UnityEngine.Random.Range(0.03f, 0.05f);

    map = new byte[worldWidth, worldHeight];

    if (IsFromSave())
    {
        //GenerateSaveFile();
        LoadMap();
        SaveMap();
    }
    else
    {
        OptimisedTerrainGeneration(map, worldWidth, worldHeight, GrassSoil, Stone);
        ApplyCaves(map);
        AddTrees(TreePopulation, TestTileFG, Log, Leaf);
    }

    Renderer(map, TestTileFG, TestTileBG);
    GenerateSaveFile();
    SaveMap();

    terrainGenerationComplete = true;
    return true;
    //map = Generate2DArray(worldWidth, worldHeight);
    //cavemap = GenerateCaveMap(worldWidth, worldHeight);
    //map = TerrainGeneration(map);
    //Renderer(map, GrassSoil, Stone, TestTileMap);
}
/// <summary>
/// Sends the map as an output.
/// </summary>
/// <returns></returns>
public byte[,] SendMap()
{

```

```

    return map;
}

//public int[,] Generate2DArray(int worldWidth, int worldHeight)
//{
//    int[,] map = new int[worldWidth, worldHeight];
//    for (int x = 0; x < worldWidth; x++)
//    {
//        for (int y = 0; y < worldHeight; y++)
//        {
//            map[x, y] = 0;
//        }
//    }
//    return map;
//}

//public int[,] GenerateCaveMap(int worldWidth, int worldHeight)
//{
//    int[,] cavemap = new int[worldWidth, worldHeight];
//    TestTexture = new Texture2D(worldWidth, worldHeight);

//    for (int x = 0; x < worldWidth; x++)
//    {
//        for (int y = 0; y < worldHeight; y++)
//        {
//            float v = Mathf.PerlinNoise((float)x *0.05f, (float)y *0.05f);
//            TestTexture.SetPixel(x, y, new Color(v, v, v));
//            if (v >= CaveThreshhold)
//            {
//                cavemap[x, y] = 0;
//            }
//            else
//            {
//                cavemap[x, y] = 1;
//            }
//            TestTexture.Apply();
//        }
//    }
//    return cavemap;
//}

///// <summary>
///// Mathf.PerlinNoise(x,y) -- Takes the X and Y coordinates and produces a smooth random
value between 0 and 1
///// </summary>
///// <param name="map"></param>
///// <returns></returns>
//public int[,] TerrainGeneration(int[,] map)
//{
//    int perlinNoise;
//    for (int x = 0; x < worldWidth; x++)
//    {
//        perlinNoise = Mathf.RoundToInt(Mathf.PerlinNoise(x / Smoothness, Seed) * worldHeight
/ 2);
//        perlinNoise += worldHeight / 2;
//        for (int y = 0; y < perlinNoise; y++)
//        {
//            map[x, y] = 1;
//        }
//    }

//    perlinNoise = 0;

```

```

//      for (int x = 0; x < worldWidth; x++)
//      {
//          perlinNoise = Mathf.RoundToInt(Mathf.PerlinNoise(x / (Smoothness*2), StoneSeed) *
worldHeight / 2);
//          perlinNoise += (worldHeight / 4);
//          for (int y = 0; y < perlinNoise; y++)
//          {
//              map[x, y] = 2;
//          }
//      }

//      return map;
//}

/// <summary>
/// Renderers the map to the tilemaps
/// </summary>
/// <param name="MapToRender"></param>
/// <param name="FG"></param>
/// <param name="BG"></param>
public void Renderer(byte[,] MapToRender, Tilemap FG, Tilemap BG)
{
    FG.ClearAllTiles();
    BG.ClearAllTiles();
    for (int x = 0; x < worldWidth; x++)
    {
        for (int y = 0; y < worldHeight; y++)
        {
            if (MapToRender[x, y] == 1)
            {
                FG.SetTile(new Vector3Int(x, y, 0), GrassSoil);
                BG.SetTile(new Vector3Int(x, y, 0), SoilBG);
            }
            if (MapToRender[x, y] == 2)
            {
                FG.SetTile(new Vector3Int(x, y, 0), Stone);
                BG.SetTile(new Vector3Int(x, y, 0), StoneBG);
            }
            if (MapToRender[x, y] == 3)
            {
                FG.SetTile(new Vector3Int(x, y, 0), Ore);
                BG.SetTile(new Vector3Int(x, y, 0), StoneBG);
            }
            if (MapToRender[x,y] == 4)
            {
                FG.SetTile(new Vector3Int(x, y, 0), Log);
            }
            if (MapToRender[x, y] == 5)
            {
                FG.SetTile(new Vector3Int(x, y, 0), Leaf);
            }
            if (MapToRender[x, y] == 6)
            {
                FG.SetTile(new Vector3Int(x, y, 0), Plank);
            }
            if (MapToRender[x,y] == 8)
            {
                BG.SetTile(new Vector3Int(x, y, 0), SoilBG);
            }
            if (MapToRender[x, y] == 9)
            {

```

```

        BG.SetTile(new Vector3Int(x, y, 0), StoneBG);
    }
}

/// <summary>
/// Generates a save file at the directory
/// </summary>
void GenerateSaveFile()
{
    if (!System.IO.File.Exists($"WorldSaves/{worldName}"))
    {
        Directory.CreateDirectory($"WorldSaves/{worldName}");
    }
}

/// <summary>
/// Checks if the world save directory already exists
/// </summary>
/// <returns></returns>
bool IsFromSave()
{
    if (System.IO.File.Exists($"WorldSaves/{worldName}")) return true;
    return false;
}

/// <summary>
/// Saves the map to a file
/// </summary>
public void SaveMap()
{
    using (StreamWriter sw = new StreamWriter($"WorldSaves/{worldName}/worldSave.txt"))
    {
        //sw.WriteLine(worldHeight);
        //sw.WriteLine(worldWidth);
        for (int y = 0; y < map.GetLength(1); y++)
        {
            for (int x = 0; x < map.GetLength(0); x++)
            {
                sw.Write(map[x, y]);
            }
            sw.WriteLine();
        }
    }
}

/// <summary>
/// Loads the map into the 2D array from a text file.
/// </summary>
public void LoadMap()
{
    int y = 0;

    using (StreamReader sr = new StreamReader($"WorldSaves/{worldName}/worldSave.txt"))
    {
        string value;

        while ((value = sr.ReadLine()) != null)
        {

```

```

        Debug.Log(Convert.ToInt16(value[1]));
        for (int x = 0; x < map.GetLength(0); x++)
        {
            map[x, y] = (byte)(Convert.ToInt16(value[x]) - 48);
        }
        y++;
    }
}
Renderer(map, TestTileFG, TestTileBG);
}

/// <summary>
/// More optimised version of the functions above using a shared for loop to reduce the
number of calls.
/// </summary>
/// <param name="worldWidth"></param>
/// <param name="worldHeight"></param>
/// <param name="GrassSoil"></param>
/// <param name="Stone"></param>
/// <param name="TestTileMap"></param>
public void OptimisedTerrainGeneration(byte[,] WorldMap, int width, int height, TileBase
GrassSoil, TileBase Stone)
{
    int perlinNoiseSoil;
    int perlinNoiseStone;
    for (int x = 0; x < width; x++)
    {
        //Making the Soil
        perlinNoiseSoil = Mathf.RoundToInt(Mathf.PerlinNoise(x / Smoothness, Seed) * height /
5);
        perlinNoiseSoil += height / 3;
        //Debug.Log($"Soil Noise Value is {perlinNoiseSoil}");
        for (int y = 0; y < perlinNoiseSoil; y++)
        {
            WorldMap[x, y] = 1;
        }

        //Making the Stone
        perlinNoiseStone = Mathf.RoundToInt(Mathf.PerlinNoise(x / (Smoothness * 2), StoneSeed)
* height / 8);
        perlinNoiseStone += height / 4;
        //Debug.Log($"Stone Noise Value is {perlinNoiseStone}");
        for (int y = 0; y < perlinNoiseStone; y++)
        {
            WorldMap[x, y] = 2;
        }

        //Cutting Caves out and Rendering to Optimise the code using the same for loops
        for (int y = 0; y < perlinNoiseSoil; y++)
        {
            float OrePL = (Mathf.PerlinNoise((float)x * OreSeed, (float)y * OreSeed));

            if (OrePL <= OreThreshold)
            {
                map[x, y] = 3;
                TestTileFG.SetTile(new Vector3Int(x, y, 0), Ore);
            }
        }
    }
}
}

```



```

    /// <summary>
    /// Made to Enable me to edit the Cave Generation without messing around with the other parts
    of the code, however this is less optimised as the loops have to be run in addition to the main
    generation.
    /// </summary>
    /// <param name="map"></param>
    /// <param name="width"></param>
    /// <param name="height"></param>
    public void ApplyCaves(byte[,] map)
    {
        //Version 1
        //for (int x = 0; x < width; x++)
        //{
        //    for (int y = 0; y < height; y++)
        //    {
        //        float CavePL = (Mathf.PerlinNoise((float)x * CaveSeed, (float)y * CaveSeed));
        //        if (CavePL >= CaveThreshold)
        //        {
        //            map[x, y] = 0;
        //            TestTileFG.SetTile(new Vector3Int(x, y, 0), null);
        //        }
        //    }
        //}

        //Version 2 With Automata
        CaveGeneration();
        Automata(iterations);

        for (int x = 0; x < worldWidth; x++)
        {
            for (int y = 0; y < worldHeight; y++)
            {
                int topofworld = Mathf.RoundToInt(Mathf.PerlinNoise(x / Smoothness, Seed) *
worldHeight / 5);
                topofworld += worldHeight / 3;
                if (cavemap[x, y] == 0 && y != 0 && y != topofworld-1 && y != topofworld-2)
                {
                    //estTileFG.SetTile(new Vector3Int(x, y, 0), null);
                    if (map[x,y] == 1)
                    {
                        map[x, y] = 8;
                    }
                    if (map[x, y] == 2 || map[x, y] == 3)
                    {
                        map[x, y] = 9;
                    }
                }
            }
        }
    }
    /// <summary>
    /// Creates a randomly filled array
    /// </summary>
    public void CaveGeneration()
    {
        cavemap = new byte[worldWidth, worldHeight];
        for (int x = 0; x < worldWidth; x++)
        {
            for (int y = 0; y < worldHeight; y++)
            {

```

```

0);
    }
}

/// <summary>
/// Runs an automata on the randomly filled grid
/// </summary>
/// <param name="count"></param>
public void Automata(int count)
{
    for (int i = 1; i <= count; i++)
    {
        byte[,] tempmap = (byte[,])cavemap.Clone();
        //Debug.Log(tempmap.GetLength(0));
        //Debug.Log(tempmap.GetLength(1));

        for (int xs = 1; xs < worldWidth - 1; xs++)
        {
            for (int ys = 1; ys < worldHeight - 1; ys++)
            {
                int neighbours = 0;
                neighbours = GetNeighbours(tempmap, xs, ys, neighbours);
                if (neighbours > 4)
                {
                    cavemap[xs, ys] = 1;
                }
                else
                {
                    cavemap[xs, ys] = 0;
                }
            }
        }
    }
}

/// <summary>
/// Returns the number of active neighbours
/// </summary>
/// <param name="tempmap"></param>
/// <param name="xs"></param>
/// <param name="ys"></param>
/// <param name="neighbours"></param>
/// <returns></returns>
private int GetNeighbours(byte[,] tempmap, int xs, int ys, int neighbours)
{
    for (int x = xs - 1; x <= xs + 1; x++)
    {
        for (int y = ys - 1; y <= ys + 1; y++)
        {
            if (y <= worldHeight && x <= worldWidth)
            {
                if (y != ys || x != xs)
                {
                    //Debug.Log(y);
                    //Debug.Log(x);
                    if (tempmap[x, y] == 1)
                    {
                        neighbours++;
                    }
                }
            }
        }
    }
}

```

```

    }
}

return neighbours;
}

/// <summary>
/// Adds trees with collision avoidance
/// </summary>
/// <param name="density"></param>
/// <param name="tilemap"></param>
/// <param name="Log"></param>
/// <param name="Leaf"></param>
public void AddTrees(int density, Tilemap tilemap, TileBase Log, TileBase Leaf)
{
    if (density >= worldWidth/6)
    {
        density = worldWidth / 6;
    }

    int tempx = 0;
    int[] positionHistory = new int[density];
    //positionHistory[0] = 0;

    for (int i = 0; i < density; i++)
    {
        bool collisionAvoidance = true;
        while (collisionAvoidance)
        {
            int holdingvariable = UnityEngine.Random.Range(2, worldWidth - 2);
            collisionAvoidance = false;

            for (int j = 0; j <= i; j++)
            {
                if (Mathf.Abs(holdingvariable - positionHistory[j]) < 6)
                {
                    //Debug.Log(Mathf.Abs(holdingvariable - positionHistory[j]));

                    collisionAvoidance = true;
                }
            }
            if (!collisionAvoidance)
            {
                tempx = holdingvariable;
            }
        }
        positionHistory[i] = tempx;

        //Debug.Log($"tempx is {tempx}");
        int tempy = worldHeight-1;
        while (map[tempx,tempy] == 0)
        {
            tempy--;
        }
        tempy++;
        //Debug.Log($"tempy is {tempy}");

        map[tempx,tempy] = 4;
        map[tempx, tempy + 1] = 4;
        map[tempx, tempy + 2] = 5;
        map[tempx, tempy + 3] = 5;
        map[tempx, tempy + 4] = 5;
    }
}

```

```

        map[tempx + 1, tempy + 2] = 5;
        map[tempx - 1, tempy + 2] = 5;
        map[tempx + 1, tempy + 3] = 5;
        map[tempx - 1, tempy + 3] = 5;
        map[tempx + 1, tempy + 4] = 5;
    }
}

/// <summary>
/// Returns a vector3 of the ground level in the middle of the world
/// </summary>
/// <returns></returns>
public Vector3Int PlayerSpawnPoint()
{
    Debug.Log("called");
    Debug.Log(worldHeight);
    Debug.Log(worldWidth);

    int tempy = worldHeight - 1;
    while (map[worldWidth/2, tempy] == 0)
    {
        tempy--;
    }
    tempy += 2;
    Debug.Log($"{worldWidth / 2}, {tempy}");
    return new Vector3Int(worldWidth / 2, tempy, 0);
}

/// <summary>
/// Unused returns the tground level of any given point
/// </summary>
/// <param name="xPos"></param>
/// <returns></returns>
public int ReturnGroundPosition(int xPos)
{
    if (xPos-1 >= worldWidth) return worldHeight;
    int tempy = worldHeight - 1;
    while (map[xPos-1, tempy] == 0)
    {
        tempy--;
    }
    tempy += 5;
    return tempy;
}

public void ClearMap()
{
    for (int x = 0; x < worldWidth; x++)
    {
        for (int y = 0; y < worldHeight; y++)
        {
            map[x, y] = 0;
        }
    }
    Renderer(map, TestTileFG, TestTileBG);
}
}

```

Runs On GameObject: WorldEventManager

Name: WorldEventManager.cs

```

using System.Collections;
using System.Collections.Generic;
using TMPPro;
using Unity.VisualScripting;
using UnityEngine;

public class WorldEventManager : MonoBehaviour
{
    public bool EnableDayNightCycle = true;
    public GameObject PlayerPrefab;
    [SerializeField] private TMP_InputField worldNameField;
    [SerializeField] private TMP_InputField PortField;
    //public GameObject MapManager;

    // Start is called before the first frame update
    void Start()
    {
        //Instantiate(PlayerPrefab,
GetComponentInChildren<GenerationScriptV2>().PlayerSpawnPoint(), Quaternion.identity);
        //Debug.Log("finito");
    }

    /// <summary>
    /// Called by the Start button on the main menu
    /// </summary>
    public void GenerateWorld()
    {
        //GetComponentInChildren<GenerationScriptV2>().SetWorldName(PassingVariables.worldName);
        //Debug.Log(PassingVariables.worldName);
        if (!string.IsNullOrEmpty(PortField.text))
        {
            GetComponentInChildren<GenerationScriptV2>().Generation(worldNameField.text);
        }
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        GetComponentInChildren<DayNightCycle>().DayNightEnabled = EnableDayNightCycle;
    }

    /// <summary>
    /// Unused
    /// </summary>
    /// <param name="Player"></param>
    public void PlayerDeath(GameObject Player)
    {
        //Player.GetComponentInChildren<PlayerMenuManager>().EnableDeathScreen();
        //Player.GetComponent<BlockInteractions>().EmptyInventory();
        //Player.transform.position = new Vector3(-1000, 0, 0);
    }

    public void PlayerRespawn(GameObject Player)
    {
        //Player.transform.position =
GetComponentInChildren<GenerationScriptV2>().PlayerSpawnPoint();
    }
}

```

```

    }

    public void SaveAll()
    {
        //GetComponentInChildren<GenerationScriptV2>().SaveMap();

        ////Finding All the Players in the scene
        //GameObject[] players;
        //players = GameObject.FindGameObjectsWithTag("Player");

        //foreach (var item in players)
        //{
        //
        item.GetComponent<BlockInteractions>().SavePlayerState(GetComponentInChildren<GenerationScriptV2>()
        ).CurrentWorldName());
        //}
    }
}

```

Runs On GameObject: Mob Spawner

Name: SpawnController.cs

```

using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;

public class SpawnController : MonoBehaviour
{
    public int spawnAmount;
    public GameObject DayNightManager;
    public GameObject WorldGenerator;
    public GameObject MobPrefab;

    [SerializeField]
    bool Day;

    GenerationScriptV2 GenerationScriptInUse;
    DayNightCycle DayNightCycleInUse;
    bool hasSpawnedMobs = false;

    // Start is called before the first frame update
    void Start()
    {
        GenerationScriptInUse = WorldGenerator.GetComponent<GenerationScriptV2>();
        DayNightCycleInUse = DayNightManager.GetComponent<DayNightCycle>();
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        Day = DayNightCycleInUse.isDay();
        if (!DayNightCycleInUse.isDay() && GenerationScriptInUse.terrainGenerationComplete &&
        !hasSpawnedMobs)
        {
            ushort ZombieID = 0;
            foreach (var positionElement in FindSpawnLocations())
            {

```

```

        FindObjectOfType<GameLogic>().CallZombieSpawn(positionElement, ZombieID);
        ZombieID++;
    }
    hasSpawnedMobs = true;
}
//if (DayNightCycleInUse.isDay() && hasSpawnedMobs)
FindObjectOfType<Zombie>().DespawnAll();
    if (DayNightCycleInUse.isDay()) hasSpawnedMobs = false;
}

/// <summary>
/// Finds valid spawn locations, and returns an array of Vector3
/// </summary>
/// <returns></returns>
Vector3[] FindSpawnLocations()
{
    Vector3[] spawnLocations = new Vector3[spawnAmount];
    for (int i = 0; i < spawnAmount; i++)
    {
        int tempxPos = UnityEngine.Random.Range(1, GenerationScriptInUse.worldWidth);
        spawnLocations[i] = new Vector3(tempxPos,
        GenerationScriptInUse.ReturnGroundPosition(tempxPos),0);
    }
    return spawnLocations;
}
}

```

Runs On GameObject: Zombie

Name: ZombieControl

```

using JetBrains.Annotations;
using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;

public class ZombieControl : MonoBehaviour
{
    public int sightRange;

    public LayerMask playerLayer;
    public LayerMask groundLayer;

    public GameObject attackPoint;
    public float attackDamage;
    public float attackRate;

    public float knockbackStrength;
    public float knockbackDuration;
    public float walkSpeed;
    public float jumpHeight = 6;
    float jumpForce;

    int ZombieHealth = 10;
    public bool OnTheGround;
    public bool InSight;
    public bool InRange;

    Rigidbody2D zombieRB;
    Animator animator;
    DayNightCycle DayNightCycleInUse;
    GameObject DayNightManager;
}

```

```

float patrolRange;
float timeSinceActive;
float time;

bool hasBeenKnockedBack;
float timeSinceKB;
float timeSinceAttack;

/// <summary>
/// These are physics overlaps, that return a boolean. If an object on the given layer is
within the given range
/// </summary>
public bool playerInRange { get { return Physics2D.OverlapCircle(transform.position,
sightRange, playerLayer); } }
public bool OnGround { get { return Physics2D.OverlapCircle(new Vector2(transform.position.x,
transform.position.y - 1.1f), 0.1f, groundLayer); } }
public bool playerInAttackRange { get { return
Physics2D.OverlapCapsule(attackPoint.transform.position, new Vector2(1, 2),
CapsuleDirection2D.Vertical, 0, playerLayer); } }

// Start is called before the first frame update
/// <summary>
/// This defines many of the components on the zombie gameobject into code
/// </summary>
void Start()
{
    zombieRB = GetComponent<Rigidbody2D>();
    animator = GetComponent<Animator>();
    DayNightManager = GameObject.Find("DayNightCycleLight");
    DayNightCycleInUse = DayNightManager.GetComponent<DayNightCycle>();
    patrolRange = UnityEngine.Random.Range(1, 20);
    RandomStartingDirection();
    Physics.IgnoreLayerCollision(gameObject.layer, gameObject.layer);
    timeSinceAttack = attackRate;
}

/// <summary>
/// Picks a random starting direction for the zombie, so when they initially patrol they all
start walking in different directions
/// </summary>
void RandomStartingDirection()
{
    int temp = Mathf.RoundToInt(UnityEngine.Random.Range(0, 1));
    if (temp == 0) transform.localScale = new Vector2(-1, 0);
    if (temp == 1) transform.localScale = new Vector2(1, 0);
}

/// <summary>
/// Used to control the behaviour of each zombie
/// </summary>
private void FixedUpdate()
{
    OnTheGround = OnGround;
    InSight = playerInRange;
    InRange = playerInAttackRange;
    //Debug.DrawLine(new Vector2(transform.position.x, transform.position.y - 1.1f), new
Vector2(transform.position.x, transform.position.y - 1.2f));
    Debug.DrawRay(new Vector2(transform.position.x, transform.position.y - 0.5f), new
Vector2(transform.localScale.x, 0), Color.red);
}

```



```

        Debug.DrawRay(new Vector2(transform.position.x, transform.position.y + 0.5f), new
Vector2(transform.localScale.x, 0), Color.red);

        ZombieWalkAnimations();
        TakeDamageDuringDay();
        JumpUpByOneBlock();
        if (playerInRange)
        {
            FaceTowardsPlayer(playerInRangePosition());
            //if (playerInAttackRange) AttackManager();
            if (!playerInAttackRange) zombieRB.velocity = new Vector2(walkSpeed *
transform.localScale.x, zombieRB.velocity.y);
        }
        else IdlePatrol();

        //if (hasBeenKnockedBack)
        //{
        //    timeSinceKB += Time.deltaTime;
        //    if (timeSinceKB >= knockbackDuration) hasBeenKnockedBack = false;
        //}

    }

    //void Attack()
    //{
    //    Collider2D[] hitPlayers = (Physics2D.OverlapCapsuleAll(attackPoint.transform.position,
new Vector2(1, 2), CapsuleDirection2D.Vertical, 0, playerLayer));
    //    //Debug.Log(hitEnemies.Length);
    //    foreach (Collider2D hits in hitPlayers)
    //    {
    //        hits.GetComponent<HealthManager>().TakeDamage((int)attackDamage);
    //    }
    //}

    //}

    //void AttackManager()
    //{
    //    timeSinceAttack += Time.deltaTime;
    //    if (timeSinceAttack > attackRate)
    //    {
    //        timeSinceAttack = 0;
    //        Attack();
    //    }
    //}

    /// <summary>
    /// Zombies local scale is changed so that they are facing a player when in range
    /// </summary>
    /// <param name="playerPos"></param>
    void FaceTowardsPlayer(Vector2 playerPos)
    {
        if (transform.position.x < playerPos.x) transform.localScale = new Vector3(1, 1, 1);
        if (transform.position.x > playerPos.x) transform.localScale = new Vector3(-1, 1, 1);
        Debug.DrawLine(transform.position, playerPos, Color.red);
    }

    /// <summary>
    /// Returns a Vector2 of the closest player within its sight range, even if there's many
    /// players in range.
    /// </summary>
    /// <returns></returns>
    public Vector2 playerInRangePosition()

```

```

{
    Vector2 closePlayer = new Vector2(0, 0);
    float previousDistance = sightRange;
    if (playerInRange)
    {
        Collider2D[] players = (Physics2D.OverlapCircleAll(transform.position, sightRange,
playerLayer));
        float currentDistance;
        foreach (var item in players)
        {
            currentDistance = (Mathf.Sqrt((float)(System.Math.Pow((item.transform.position.x -
transform.position.x), 2) + System.Math.Pow((item.transform.position.y - transform.position.y),
2))));
            if (currentDistance < previousDistance)
            {
                closePlayer = item.transform.position;
                previousDistance = currentDistance;
            }
        }
        return closePlayer;
    }

    /// <summary>
    /// The zombies will walk in an alternating direction for a random amount of time, as defined
in Start()
    /// </summary>
    void IdlePatrol()
    {
        float lengthOfTime = walkSpeed * patrolRange;
        if ((patrolRange > 0) && !playerInRange)
        {
            FaceTowardsWalkingDirection();
            timeSinceActive += Time.deltaTime;
            if (timeSinceActive > 2 * lengthOfTime)
            {
                timeSinceActive -= 2 * lengthOfTime;
            }

            if (timeSinceActive > lengthOfTime)
            {
                time = timeSinceActive - lengthOfTime;
                zombieRB.velocity = new Vector2(walkSpeed * transform.localScale.x,
zombieRB.velocity.y);
            }
            else
            {
                time = timeSinceActive;
                zombieRB.velocity = new Vector2(walkSpeed * transform.localScale.x,
zombieRB.velocity.y);
            }
        }
        timeSinceActive = 0;
    }

    /// <summary>
    /// If the zombie detects a block infront of itself that is only one block tall, it will jump
over it.
    /// </summary>
    void JumpUpByOneBlock()

```

```

{
    if (Physics2D.Raycast(new Vector2(transform.position.x, transform.position.y - 0.5f), new
Vector2(transform.localScale.x, 0), 0.5f, groundLayer) && !Physics2D.Raycast(new
Vector2(transform.position.x, transform.position.y + 0.5f), new Vector2(transform.localScale.x,
0), 0.5f, groundLayer) && OnGround)
    {
        jumpForce = Mathf.Sqrt(jumpHeight * -2 * (Physics2D.gravity.y *
zombieRB.gravityScale));
        zombieRB.AddForce(new Vector3(0.0f, jumpForce, 0.0f), ForceMode2D.Impulse);
        //Debug.Log($"Jump Called, {jumpForce}");
    }

}

/// <summary>
/// Faces the zombie towards its velocity.
/// </summary>
void FaceTowardsWalkingDirection()
{
    if (zombieRB.velocity.x > 0)
    {
        transform.localScale = new Vector3(1, 1, 1);
    }
    if (zombieRB.velocity.x < 0)
    {
        transform.localScale = new Vector3(-1, 1, 1);
    }
}

/// <summary>
/// Despawns during the daytime
/// </summary>
void TakeDamageDuringDay()
{
    if (DayNightCycleInUse.isDay())
    {
        GetComponent<Zombie>().Despawn();
    }
}

/// <summary>
/// Simple animation trigger
/// </summary>
void ZombieWalkAnimations()
{
    if (zombieRB.velocity.x != 0)
    {
        animator.Play("ZombieWalk");
    }
    else animator.Play("ZombieIdle");
}

/// <summary>
/// Unused on the server side
/// </summary>
/// <param name="damage"></param>
/// <param name="isFromPlayer"></param>
public void TakeDamage(int damage, bool isFromPlayer)
{
    if (isFromPlayer)
    {
        FindObjectOfType<AudioManager>().Play("Mob Damage");
    }
}

```

```

    }
    if (ZombieHealth - damage > 0)
    {
        ZombieHealth -= damage;
    }
    else Destroy(gameObject);
}

//public void ApplyKnockback(Vector2 attackPos)
//{
//    hasBeenKnockedBack = true;
//    timeSinceKB = 0;
//    Vector2 direction = ((Vector2)transform.position - attackPos).normalized;
//    zombieRB.AddForce(direction * knockbackStrength, ForceMode2D.Impulse);
//    Debug.Log(hasBeenKnockedBack);
//}
}

```

Runs On GameObject: DayNightManager

Name: DayNightCycle.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DayNightCycle : MonoBehaviour
{
    public float MoveY;
    public float DayOrNightTime;

    Vector3 StartPos;
    float timeSinceActive;
    float time;
    public bool DayNightEnabled { get; set; }

    void Start()
    {
        StartPos = transform.position;
    }

    // Days will be 5 mins, Nights will be 5 mins
    void FixedUpdate()
    {
        if (FindObjectOfType<NetworkManager>().Server.IsRunning) MovingTheLight();
    }

    /// <summary>
    /// Moves the light source within the scene in a periodic vertical wave.
    /// </summary>
    void MovingTheLight()
    {
        if ((MoveY > 0) && DayOrNightTime > 0 && DayNightEnabled)
        {
            timeSinceActive += Time.deltaTime;

```

```

        if (timeSinceActive > 2 * DayOrNightTime)
        {
            timeSinceActive -= 2 * DayOrNightTime;
        }

        if (timeSinceActive > DayOrNightTime)
        {
            time = timeSinceActive - DayOrNightTime;
            transform.position = new Vector3(StartPos.x, (StartPos.y + MoveY) - MoveY * (time
/ DayOrNightTime), StartPos.z);
        }
        else
        {
            time = timeSinceActive;
            transform.position = new Vector3(StartPos.x, StartPos.y + MoveY * (time /
DayOrNightTime), StartPos.z);
        }
    }
}

/// <summary>
/// Returns true if the light is far enough away for the ground to be dark.
/// </summary>
/// <returns></returns>
public bool isDay()
{
    if (timeSinceActive >= DayOrNightTime * 0.5 && timeSinceActive <= DayOrNightTime * 1.5)
return false;
    else return true;
}

/// <summary>
/// Returns the position of the light
/// </summary>
/// <returns></returns>
public Vector3 ReturnLightPosition() { return transform.position; }
}

```

Client Scripts

Runs On GameObject: NetworkManager

Name: NetworkManager.cs

```
using RiptideNetworking;
using RiptideNetworking.Utls;
using UnityEngine;
using System;
using TMPro;
//using UnityEditor.Experimental.GraphView;

public enum ServerToClientId : ushort
{
    playerSpawned = 1,
    map,
    syncNonLocalPosition,
    syncMapUpdate,
    lightPosition,
    zombieSpawning,
    zombiePosition,
    zombieDeath,
    textChat,
    despawnPlayer,
}

public enum ClientToServerId : ushort
{
    name = 1,
    updatePlayerPosition,
    updateServerMap,
    zombieDeath,
    updateTextChat,
}

public class NetworkManager : MonoBehaviour
{
    private static NetworkManager instance;

    public static NetworkManager Instance
    {
        get => instance;
        private set
        {
            if (instance == null) instance = value;
            else if (instance != value)
            {
                Debug.Log($"{nameof(NetworkManager)} instance already exists, destroying new");
                Destroy(value);
            }
        }
    }

    public Client Client { get; private set; }
    public TextChatManger DebugText;
    //[SerializeField] private string ip;
    //[SerializeField] private ushort port;

    private void Awake()
    {

```

```

        instance = this;
        Application.runInBackground = true;
    }

    /// <summary>
    /// Functions Added to the Clients methods that are called ontop of the normal methods,
    similar to overrides
    /// </summary>
    private void Start()
    {
        RiptideLogger.Initialize(Debug.Log, Debug.Log, Debug.LogWarning, Debug.LogError, false);
        Client = new Client();

        Client.Connected += DidConnect;
        Client.ConnectionFailed += FailedToConnect;
        Client.Disconnected += DidDisconnect;
        Client.ClientDisconnected += PlayerLeft;
        //Client.Disconnect += CalledLeave;
    }

    ///
    private void FixedUpdate()
    {
        Client.Tick();
    }

    private void OnApplicationQuit()
    {
        Client.Disconnect();
    }

    public void Connect(string ip, ushort port)
    {
        //DebugText.AddToChat($"Attempting Connection on {ip}:{port}, at
{Time.realtimeSinceStartup}");
        Client.Connect($"{ip}:{port}");
    }

    private void PlayerLeft(object sender, ClientDisconnectedEventArgs e)
    {
        Destroy(Player.list[e.Id]);
    }

    private void DidConnect(object sender, EventArgs e)
    {
        UIManager.Instance.SendName();
        DebugText.AddToChat($"Connected at {Time.realtimeSinceStartup}");
    }

    private void FailedToConnect(object sender, EventArgs e)
    {
        DebugText.AddToChat($"Failed To Connect at {Time.realtimeSinceStartup}");
        UIManager.Instance.BackToMain();
    }

    /// <summary>
    /// If a client diconnects all Local and Non-Local players need to be destroyed
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>

```

```

private void DidDisconnect(object sender, EventArgs e)
{
    DebugText.AddToChat($"Disconnected at {Time.realtimeSinceStartup}");
    UIManager.Instance.BackToMain();
    RemoveEntities();
    UIManager.Instance.BackToMain();
}

public void CalledLeave()
{
    Client.Disconnect();
    DebugText.AddToChat($"Player Called Left at {Time.realtimeSinceStartup}");
    Debug.Log("Disconnect Called");
    RemoveEntities();
    UIManager.Instance.BackToMain();
}

void RemoveEntities()
{
    foreach (Player players in Player.list.Values)
    {
        Destroy(players.gameObject);
    }
    foreach (Zombie item in Zombie.list.Values)
    {
        Destroy(item.gameObject);
    }
}
}

```

Runs On GameObject: N/A //From to RiptideNetworking GitHub

Name: MessageExtension.cs

```

using RiptideNetworking;
using UnityEngine;

public static class MessageExtensions
{
    #region Vector2
    /// <inheritdoc cref="Add(Message, Vector2)"/>
    /// <remarks>Relying on the correct Add overload being chosen based on the parameter type can
    increase the odds of accidental type mismatches when retrieving data from a message. This method
    calls <see cref="Add(Message, Vector2)"/> and simply provides an alternative type-explicit way to
    add a <see cref="Vector2"/> to the message.</remarks>
    public static Message AddVector2(this Message message, Vector2 value) => Add(message, value);

    /// <summary>Adds a <see cref="Vector2"/> to the message.</summary>
    /// <param name="value">The <see cref="Vector2"/> to add.</param>
    /// <returns>The message that the <see cref="Vector2"/> was added to.</returns>
    public static Message Add(this Message message, Vector2 value)
    {
        message.AddFloat(value.x);
        message.AddFloat(value.y);
        return message;
    }

    /// <summary>Retrieves a <see cref="Vector2"/> from the message.</summary>
    /// <returns>The <see cref="Vector2"/> that was retrieved.</returns>

```



```

public static Vector2 GetVector2(this Message message)
{
    return new Vector2(message.GetFloat(), message.GetFloat());
}
#endregion

#region Vector3
/// <inheritdoc cref="Add(Message, Vector3)"/>
/// <remarks>Relying on the correct Add overload being chosen based on the parameter type can
increase the odds of accidental type mismatches when retrieving data from a message. This method
calls <see cref="Add(Message, Vector3)"/> and simply provides an alternative type-explicit way to
add a <see cref="Vector3"/> to the message.</remarks>
public static Message AddVector3(this Message message, Vector3 value) => Add(message, value);

/// <summary>Adds a <see cref="Vector3"/> to the message.</summary>
/// <param name="value">The <see cref="Vector3"/> to add.</param>
/// <returns>The message that the <see cref="Vector3"/> was added to.</returns>
public static Message Add(this Message message, Vector3 value)
{
    message.AddFloat(value.x);
    message.AddFloat(value.y);
    message.AddFloat(value.z);
    return message;
}

/// <summary>Retrieves a <see cref="Vector3"/> from the message.</summary>
/// <returns>The <see cref="Vector3"/> that was retrieved.</returns>
public static Vector3 GetVector3(this Message message)
{
    return new Vector3(message.GetFloat(), message.GetFloat(), message.GetFloat());
}
#endregion

#region Quaternion
/// <inheritdoc cref="Add(Message, Quaternion)"/>
/// <remarks>Relying on the correct Add overload being chosen based on the parameter type can
increase the odds of accidental type mismatches when retrieving data from a message. This method
calls <see cref="Add(Message, Quaternion)"/> and simply provides an alternative type-explicit way
to add a <see cref="Quaternion"/> to the message.</remarks>
public static Message AddQuaternion(this Message message, Quaternion value) => Add(message,
value);

/// <summary>Adds a <see cref="Quaternion"/> to the message.</summary>
/// <param name="value">The <see cref="Quaternion"/> to add.</param>
/// <returns>The message that the <see cref="Quaternion"/> was added to.</returns>
public static Message Add(this Message message, Quaternion value)
{
    message.AddFloat(value.x);
    message.AddFloat(value.y);
    message.AddFloat(value.z);
    message.AddFloat(value.w);
    return message;
}

/// <summary>Retrieves a <see cref="Quaternion"/> from the message.</summary>
/// <returns>The <see cref="Quaternion"/> that was retrieved.</returns>
public static Quaternion GetQuaternion(this Message message)
{
    return new Quaternion(message.GetFloat(), message.GetFloat(), message.GetFloat(),
message.GetFloat());
}
#endregion
}

```

Runs On GameObject: NetworkManager

Name: GameLogic.cs

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

public class GameLogic : MonoBehaviour
{
    private static GameLogic instance;
    private static GameObject worldLight;

    /// <summary>
    /// Creates a Singleton
    /// </summary>
    public static GameLogic Instance
    {
        get => instance;
        private set
        {
            if (instance == null) instance = value;
            else if (instance != value)
            {
                Debug.Log($"{nameof(GameLogic)} instance already exists, destroying new");
                Destroy(value);
            }
        }
    }

    /// <summary>
    /// Called before the object is loaded
    /// </summary>
    private void Awake()
    {
        instance = this;
        worldLight = FindObjectOfType<DayNightCycle>().gameObject;
    }

    /// <summary>
    /// Enable indirect access to the gameobjects, for security reasons
    /// </summary>
    public GameObject LocalPlayerPrefab => localPlayerPrefab;
    public GameObject PlayerPrefab => playerPrefab;
    //public GameObject ZombiePrefab => zombiePrefab;

    /// <summary>
```

```

    /// Serialised Fields are used to make the variables show up in the Unity Editor while
    remaining private
    /// </summary>
    [Header("Prefabs")]
    [SerializeField] private GameObject localPlayerPrefab;
    [SerializeField] private GameObject playerPrefab;

    /// <summary>
    /// Message Handler for moving the light source within the scene
    /// </summary>
    /// <param name="message"></param>
    [MessageHandler((ushort)ServerToClientId.lightPosition)]
    private static void ClientLightPosition(Message message)
    {
        worldLight.transform.position = (message.GetVector3());
        //Debug.Log("light pos received");
    }

    /// <summary>
    /// Enable indirect access to the gameobjects, for security reasons
    /// </summary>
    public GameObject ZombiePrefab => zombiePrefab;

    [SerializeField] private GameObject zombiePrefab;
}

```

Runs On GameObject: UI Canvas

Name: UIManager.cs

```

using RiptideNetworking;
using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.UI;
using TMPro;
//using UnityEditor.Experimental.GraphView;
using System;

public class UIManager : MonoBehaviour
{
    private static UIManager instance;

    /// <summary>
    /// Creates a Singleton, to ensure that only one version of this class is used
    /// </summary>
    public static UIManager Instance
    {
        get => instance;
        private set
        {
            if (instance == null) instance = value;
            else if (instance != value)
            {
                Debug.Log($"{nameof(UIManager)} instance already exists, destroying new");
                Destroy(value);
            }
        }
    }
}

```

```

    }
}

[Header("Connect")]
[SerializeField] private GameObject connectUI;
[SerializeField] private TMP_InputField usernameField;
[SerializeField] private TMP_InputField ipField;
[SerializeField] private TMP_InputField portField;

/// <summary>
/// Is run before the object is loaded into the scene
/// </summary>
private void Awake()
{
    instance = this;
}

/// <summary>
/// Accessible by the connect button, checks for IP and Port fields not being valid
/// </summary>
public void ConnectClicked()
{
    if (portField.text.Length == 4 && !String.IsNullOrEmpty(ipField.text))
    {
        usernameField.interactable = false;
        connectUI.SetActive(false);
        NetworkManager.Instance.DebugText.AddToChat($"Attempting Connection on
{ipField.text}:{System.Convert.ToUInt16(portField.text)}, at {Time.realtimeSinceStartup}");
        NetworkManager.Instance.Connect(ipField.text,
System.Convert.ToUInt16(portField.text));
    }
}

/// <summary>
/// Used to return the user to the main menu, if they are disconnected or fail to join
/// </summary>
public void BackToMain()
{
    usernameField.interactable = true;
    connectUI.SetActive(true);
}

/// <summary>
/// Sends the username of as new client to the Server
/// </summary>
public void SendName()
{
    RiptideNetworking.Message message = Message.Create(MessageSendMode.reliable,
(ushort)ClientId.name);
    message.AddString(usernameField.text);
    NetworkManager.Instance.Client.Send(message);
}
}

```

Runs On GameObject: Local and Non-Local Player

Name: Player.cs

```

using System.Collections;
using RiptideNetworking;
using RiptideNetworking.Utills;
using System.Collections.Generic;
using UnityEngine;
using Unity.VisualScripting;
using TMPro;

public class Player : MonoBehaviour
{
    public static Dictionary<ushort, Player> list = new Dictionary<ushort, Player>();

    public ushort Id { get; private set; }
    public bool IsLocal { get; private set; }
    public string userName { get; private set; }

    [SerializeField] private GenerationScriptV2 Generation;

    private void OnDestroy()
    {
        list.Remove(Id);
    }

    /// <summary>
    /// Runs 60 time per second
    /// </summary>
    private void FixedUpdate()
    {
        PassingPlayerPositionToServer();
    }

    /// <summary>
    /// Sends the Vector3 of the players position to the server
    /// </summary>
    private void PassingPlayerPositionToServer()
    {
        if (Id == NetworkManager.Instance.Client.Id)
        {
            //Debug.Log("positon Update called");
            RiptideNetworking.Message message = Message.Create(MessageSendMode.unreliable,
(ushort)ClientToServerId.updatePlayerPosition);
message.AddVector3(list[NetworkManager.Instance.Client.Id].gameObject.transform.position);
            //message.AddUShort(Id);
            NetworkManager.Instance.Client.Send(message);
        }
    }

    /// <summary>
    /// Called by the server, Adds the player to the Dictionary along with filling out variables.
    /// Also spawns in a gameObject for Local and Non-Local players.
    /// </summary>
    /// <param name="id"></param>
    /// <param name="username"></param>
    /// <param name="position"></param>
    public static void Spawn(ushort id, string username, Vector3 position)
    {
        Player player;
        if (id == NetworkManager.Instance.Client.Id)
        {
            player = Instantiate(GameLogic.Instance.LocalPlayerPrefab, position,
Quaternion.identity).GetComponent<Player>();
            player.IsLocal = true;

```

```

        player.gameObject.GetComponent<usernameText>().ApplyUserName(username);
    }
    else
    {
        player = Instantiate(GameLogic.Instance.PlayerPrefab, position,
Quaternion.identity).GetComponent<Player>();
        player.IsLocal = false;
        player.gameObject.GetComponent<usernameText>().ApplyUserName(username);
    }
    player.name = $"Player {id} ({(string.IsNullOrEmpty(username) ? "Guest" : username)})";
    player.Id = id;
    player.userName = username;
    list.Add(id, player);
}

//private static void CallMapBuild(int xlength, int ylength, int[] mapAs1DArray)
//{
//    int[,] map = new int[xlength, ylength];
//    //int tempy = 0;
//    for (int y = 0; y < ylength; y++)
//    {
//        for (int x = 0; x < xlength; x++)
//        {
//            map[x, y] = mapAs1DArray[x];
//        }
//    }
//    FindObjectOfType<GenerationScriptV2>().Renderer(map);
//    FindObjectOfType<GenerationScriptV2>().setMap(map);
//}

/// <summary>
/// Sends an update request to the server for the text chat, upon this client adding to it
/// </summary>
/// <param name="text"></param>
public void UpdateTextChat(string text)
{
    Message message = Message.Create(MessageSendMode.reliable,
(ushort)ClientId.updateTextChat);
    message.AddString(text);
    NetworkManager.Instance.Client.Send(message);
}

/// <summary>
/// Called from within the build/break block function by the client to inform the server a
block has been altered.
/// </summary>
/// <param name="x"></param>
/// <param name="y"></param>
/// <param name="block"></param>
public void SendBlockUpdateToServer(int x, int y, byte block)
{
    RiptideNetworking.Message message = Message.Create(MessageSendMode.reliable,
(ushort)ClientId.updateServerMap);
    message.AddInt(x);
    message.AddInt(y);
    message.AddByte(block);
    NetworkManager.Instance.Client.Send(message);
}

/// <summary>
/// Called from a message handler below to fill in a layer of the map.
/// </summary>
/// <param name="mapSlice"></param>

```

```

/// <param name="mapLayer"></param>
private static void CallMapBuild(byte[] mapSlice, short mapLayer)
{
    FindObjectOfType<GenerationScriptV2>().setMapByLayer(mapSlice, mapLayer);
}

/// <summary>
/// Message sent from the server to spawn a player.
/// </summary>
/// <param name="message"></param>
[MessageHandler((ushort)(ServerToClientId.playerSpawned))]
private static void SpawnPlayer(Message message)
{
    Spawn(message.GetUShort(), message.GetString(), message.GetVector3());
}

/// <summary>
/// Message handler for recieving layers of the map from the server.
/// </summary>
/// <param name="message"></param>
[MessageHandler((ushort)(ServerToClientId.map))]
private static void GettingMapFromServer(Message message)
{
    //
    short Layer = message.GetShort();
    int lengthofByteArray = message.GetInt();
    byte[] ByteArray = new byte[lengthofByteArray];
    message.GetBytes(lengthofByteArray, ByteArray);
    //short LayerInteger = message.GetShort();
    //Debug.Log(LayerInteger);
    //Debug.LogAssertion(Layer);
    //Debug.Log("Received Call From Server To Build Map");

    CallMapBuild(ByteArray, Layer);
    message.Release();
}

/// <summary>
/// Syncs the position on Non-Local player to the position provided by the server.
/// </summary>
/// <param name="message"></param>
[MessageHandler((ushort)(ServerToClientId.syncNonLocalPosition))]
private static void SyncingPlayers(Message message)
{
    ushort playerID = message.GetUShort();
    Vector3 playerPosition = message.GetVector3();
    if (playerID != NetworkManager.Instance.Client.Id)
    {
        Player.list[playerID].gameObject.transform.position = playerPosition;
    }
    message.Release();
}

/// <summary>
/// Updates the Client's map from any changes to the servers map, made by other clients
connected
/// </summary>
/// <param name="message"></param>
[MessageHandler((ushort)(ServerToClientId.syncMapUpdate))]
private static void SyncingMaps(Message message)

```

```

{
    int xPos = message.GetInt();
    int yPos = message.GetInt();
    byte block = message.GetByte();
    //Debug.Log($"Block Update Called at {(int)BlockPos.x}:{(int)BlockPos.y} for {block}");

FindObjectOfType<WorldEventManager>().GetComponentInChildren<GenerationScriptV2>().ServerUpdatingBlock(block, xPos, yPos);
    message.Release();
    //UpdatePlayerMaps(block, xPos, yPos, fromClientId);

}

/// <summary>
/// Syncs the text chat if any other client has added to it
/// </summary>
/// <param name="message"></param>
[MessageHandler((ushort)ServerToClientId.textChat)]
private static void SyncingTextChat(Message message)
{
    if (message.GetUShort() == NetworkManager.Instance.Client.Id)
    {
        FindObjectOfType<TextChatManger>().AddToChat(message.GetString());
    }

    message.Release();
}

[MessageHandler((ushort)ServerToClientId.despawnPlayer)]
private static void DespawnDisconnectedPlayer(Message message)
{
    Destroy(list[message.GetUShort()].gameObject);
}
}

```

Runs On GameObject: Zombie

Name: Zombie.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using RiptideNetworking;
using RiptideNetworking.Utls;

public class Zombie : MonoBehaviour
{
    /// <summary>
    /// Dictionary that stores the Class and Id of each zombie
    /// </summary>
    public static Dictionary<ushort, Zombie> list = new Dictionary<ushort, Zombie>();

    public ushort Id { get; private set; }

    private void OnDestroy()
    {
        list.Remove(Id);
    }

    /// <summary>

```



```

    /// Requests to despawn zombies from the server, which will then remove them and tell the
    client.
    /// </summary>
    public void DespawnFromClient()
    {
        Message message = Message.Create(MessageSendMode.reliable,
(ushort)ClientToServerId.zombieDeath);
        message.AddUShort(Id);
        NetworkManager.Instance.Client.Send(message);
        //Destroy(list[Id].gameObject);
    }

    /// <summary>
    /// Message Handler for spawning a zombie, at a given position and with Id.
    /// </summary>
    /// <param name="message"></param>
    [MessageHandler((ushort)ServerToClientId.zombieSpawning)]
    private static void Spawn(Message message)
    {
        Zombie zombie = Instantiate(GameLogic.Instance.ZombiePrefab, message.GetVector3(),
Quaternion.identity).GetComponent<Zombie>();
        zombie.Id = message.GetUShort();
        list.Add(zombie.Id, zombie);
    }

    /// <summary>
    /// Message handler for updating the position of a zombie from the server
    /// </summary>
    /// <param name="message"></param>
    [MessageHandler((ushort)ServerToClientId.zombiePosition)]
    private static void UpdatingZombiePosition(Message message)
    {
        list[message.GetUShort()].gameObject.transform.position = message.GetVector3();
    }

    /// <summary>
    /// Update the dictionary, with any zombie despawns requested by the server
    /// </summary>
    /// <param name="message"></param>
    [MessageHandler((ushort)ServerToClientId.zombieDeath)]
    private static void DespawnZombie(Message message)
    {
        ushort idOfDeath = message.GetUShort();
        Destroy(list[idOfDeath].gameObject);
        message.Release();
    }
}

```

Runs On GameObject: GeneratorV2

Name: GenerationScriptV2.cs

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Text;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.Tilemaps;
using UnityEngine.UI;

public class GenerationScriptV2 : MonoBehaviour
{
    // Start is called before the first frame update
    public int worldWidth = 1024;
    public int worldHeight = 256;
    public bool terrainGenerationComplete { get; private set; }

    [SerializeField] float Seed;
    [SerializeField] float StoneSeed;
    //[SerializeField] float CaveSeed;
    [SerializeField] float OreSeed;

    float Smoothness = 40;
    float OreThreshold = 0.1f;
    //float CaveThreshold = 0.7f;

    [SerializeField] TileBase GrassSoil;
    [SerializeField] TileBase Stone;
    [SerializeField] TileBase SoilBG;
    [SerializeField] TileBase StoneBG;
    [SerializeField] TileBase Ore;
    [SerializeField] TileBase Log;
    [SerializeField] TileBase Leaf;
    [SerializeField] TileBase Plank;

    [SerializeField] Tilemap TestTileFG;
    [SerializeField] Tilemap TestTileBG;

    [SerializeField] public Slider SeedSlider;
    [SerializeField] public Slider StoneSlider;
    //[SerializeField] public Slider CaveSlider;

    [SerializeField] float RandomPercentFill;
    [SerializeField] int iterations;
    [SerializeField] int TreePopulation;

    string worldName;

    byte[,] map;
    byte[,] cavemap;
    //int layer;

    void Start()
    {
        worldWidth = 1024;
        worldHeight = 256;
        map = new byte[worldWidth, worldHeight];

        //layer = 0;
    }
}

```

```

    //terrainGenerationComplete = Generation();
}

//void Update()
//{
//    if (Input.GetKeyDown(KeyCode.Space))
//    {
//        Debug.Log("Reloading");
//        Generation();
//    }
//}
public void SetWorldName(string name) { worldName = name; }

/// <summary>
/// Returns the block broken to the player.
/// </summary>
/// <param name="x"></param>
/// <param name="y"></param>
/// <returns></returns>
public int BreakBlock(int x, int y)
{
    //Debug.Log($"breakblock called, position {x}, {y}");
    if (x >= worldWidth || y >= worldHeight) return 0;

    if (!(map[x, y] == 8 || map[x, y] == 9 || map[x, y] == 0))
    {
        TestTileFG.SetTile(new Vector3Int(x, y, 0), null);
        TestTileFG.RefreshTile(new Vector3Int(x, y, 0));
        int block = map[x, y];
        if (map[x, y] == 1)
        {
            map[x, y] = 8;
        }
        else if (map[x, y] == 2 || map[x, y] == 3)
        {
            map[x, y] = 9;
        }
        else map[x, y] = 0;
        FindObjectOfType<AudioManager>().Play("Block Break");
        FindObjectOfType<Player>().SendBlockUpdateToServer(x, y, map[x, y]);
        return block;
    }
    return 0;
}

/// <summary>
/// Used to update the map without checks as this is called by the server and has already been
check on the other clients end.
/// </summary>
/// <param name="block"></param>
/// <param name="x"></param>
/// <param name="y"></param>
public void ServerUpdatingBlock(int block, int x, int y)
{
    if (x >= worldWidth || y >= worldHeight) return;
    map[x, y] = (byte)block;
    TileBase placing = Plank;
    switch (block)
    {
        case 0:
            placing = null;
            break;
    }
}

```

```

        case 1:
            placing = GrassSoil;
            break;
        case 2:
            placing = Stone;
            break;
        case 4:
            placing = Log;
            break;
        case 5:
            placing = Leaf;
            break;
        case 6:
            placing = Plank;
            break;
        case 8:
            placing = SoilBG;
            break;
        case 9:
            placing = StoneBG;
            break;
        default:
            break;
    }
    if (block == 8 || block == 9)
    {
        TestTileBG.SetTile(new Vector3Int(x, y, 0), placing);
        TestTileFG.SetTile(new Vector3Int(x, y, 0), null);
        TestTileBG.RefreshTile(new Vector3Int(x, y, 0));
        TestTileFG.RefreshTile(new Vector3Int(x, y, 0));
    }
    else
    {
        TestTileFG.SetTile(new Vector3Int(x, y, 0), placing);
        TestTileFG.RefreshTile(new Vector3Int(x, y, 0));
        TestTileBG.RefreshTile(new Vector3Int(x, y, 0));
    }

    //Renderer(map, TestTileFG, TestTileBG);
    //Debug.Log("GenerationScript Called");
}

/// <summary>
/// Used in TimeBreaking() to find the amount of time it takes to break a block. Also used to
add to teh players inventory
/// </summary>
/// <param name="x"></param>
/// <param name="y"></param>
/// <returns></returns>
public int ReturnTypeOfBlock(int x, int y)
{
    if (x >= worldWidth || y >= worldHeight) return 0;
    if (!(map[x, y] == 8 || map[x, y] == 9 || map[x, y] == 0))
    {
        int block = map[x, y];
        return block;
    }
    return 0;
}

/// <summary>

```

```

/// Builds a block at the given position of given type. Check that the placement is valid
/// </summary>
/// <param name="block"></param>
/// <param name="x"></param>
/// <param name="y"></param>
/// <returns></returns>
public bool BuildBlock(int block,int x, int y)
{
    if (x >= worldWidth || y >= worldHeight) return false;
    //Debug.Log($"buildblock called, position {x}, {y}");
    if ((map[x-1,y] != 0 || map[x + 1, y] != 0 || map[x, y - 1] != 0 || map[x, y + 1] != 0) &&
(map[x,y] == 0 || map[x, y] == 8 || map[x, y] == 9))
    {
        map[x, y] = (byte)block;
        TileBase placing = Plank;
        switch (block)
        {
            case 1:
                placing = GrassSoil;
                break;
            case 2:
                placing = Stone;
                break;
            case 4:
                placing = Log;
                break;
            case 5:
                placing = Leaf;
                break;
            case 6:
                placing = Plank;
                break;
            default:
                return false;
        }

        TestTileFG.SetTile(new Vector3Int(x, y, 0), placing);
        TestTileFG.RefreshTile(new Vector3Int(x, y, 0));
        FindObjectOfType<AudioManager>().Play("Block Place");
        FindObjectOfType<Player>().SendBlockUpdateToServer(x, y, map[x, y]);
        return true;
    }
    return false;
}

/// <summary>
/// Adds to teh 2D map array layer by layer, this is done as the capacity for mesages is 1200
byte, therefore the map must be sent layer by layer in order to fit.
/// </summary>
/// <param name="newmap"></param>
/// <param name="layer"></param>
public void setMapByLayer(byte[] newmap, short layer)
{
    Debug.Log($"layer is {layer}");
    for (int x = 0; x < map.GetLength(0); x++)
    {
        map[x, layer] = newmap[x];
    }
}

```

```

    if (layer == worldHeight - 1)
    {
        Renderer(map);
    }
    //layer++;
}

/// <summary>
/// Renders teh content of the 2D array to the Tilemaps
/// </summary>
/// <param name="MapToRender"></param>
public void Renderer(byte[,] MapToRender)
{
    TestTileFG.ClearAllTiles();
    TestTileBG.ClearAllTiles();
    for (int x = 0; x < worldWidth; x++)
    {
        for (int y = 0; y < worldHeight; y++)
        {
            if (MapToRender[x, y] == 1)
            {
                TestTileFG.SetTile(new Vector3Int(x, y, 0), GrassSoil);
                TestTileBG.SetTile(new Vector3Int(x, y, 0), SoilBG);
            }
            if (MapToRender[x, y] == 2)
            {
                TestTileFG.SetTile(new Vector3Int(x, y, 0), Stone);
                TestTileBG.SetTile(new Vector3Int(x, y, 0), StoneBG);
            }
            if (MapToRender[x, y] == 3)
            {
                TestTileFG.SetTile(new Vector3Int(x, y, 0), Ore);
                TestTileBG.SetTile(new Vector3Int(x, y, 0), StoneBG);
            }
            if (MapToRender[x,y] == 4)
            {
                TestTileFG.SetTile(new Vector3Int(x, y, 0), Log);
            }
            if (MapToRender[x, y] == 5)
            {
                TestTileFG.SetTile(new Vector3Int(x, y, 0), Leaf);
            }
            if (MapToRender[x, y] == 6)
            {
                TestTileFG.SetTile(new Vector3Int(x, y, 0), Plank);
            }
            if (MapToRender[x,y] == 8)
            {
                TestTileBG.SetTile(new Vector3Int(x, y, 0), SoilBG);
            }
            if (MapToRender[x, y] == 9)
            {
                TestTileBG.SetTile(new Vector3Int(x, y, 0), StoneBG);
            }
        }
    }
}

public byte[,] sendMap()
{

```

```

        return map;
    }

    public Vector3Int PlayerSpawnPoint()
    {
        Debug.Log("called");
        Debug.Log(worldHeight);
        Debug.Log(worldWidth);

        int tempy = worldHeight - 1;
        while (map[worldWidth/2, tempy] == 0)
        {
            tempy--;
        }
        tempy += 2;
        Debug.Log($"{worldWidth / 2}, {tempy}");
        return new Vector3Int(worldWidth / 2, tempy, 0);
    }
}

```

Runs On GameObject: LocalPlayer

Name: PlayerController.cs

```

using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public float moveX;
    public float speed;
    public float jumpHeight;
    public bool OnTheGround;
    public float checkradius;
    public CapsuleCollider2D PlayerCC;
    public Rigidbody2D PlayerRB;
    public LayerMask groundLayer;
    Animator animator;

    float jumpForce;
    bool facingRight = true;

    bool isFalling { get { return (!OnTheGround && PlayerRB.velocity.y < 0); } }
    bool wasFalling;
    bool wasGrounded;
    float startofFall;

    // Start is called before the first frame update

```

```

void Start()
{
    animator = GetComponent<Animator>();
}

// Update is called once per frame
/// <summary>
/// Takes in the users inputs to control the player
/// </summary>
void Update()
{
    if (GetComponentInChildren<PlayerMenuManager>().hasAnyMenuOpen) return;
    xMovePlayer();
    PlayerWalkAnimation();
    //PlayerFall();

    jumpForce = Mathf.Sqrt(jumpHeight * -2 * (Physics2D.gravity.y * PlayerRB.gravityScale));
    if (Input.GetKey(KeyCode.Space) && OnGround())
    {
        PlayerRB.AddForce(new Vector3(0.0f, jumpForce, 0.0f), ForceMode2D.Impulse);
    }

    //Debug.Log(PlayerRB.velocity.x);
}

/// <summary>
/// Control the players movement on the x-axis
/// </summary>
void xMovePlayer()
{
    PlayerDirection();
    moveX = Input.GetAxis("Horizontal");
    Vector2 movement = new Vector2(moveX, 0);
    if (OnGround()) PlayerRB.velocity = movement * speed;
    else if (!OnGround() && moveX != 0) PlayerRB.AddForce(new Vector2(transform.localScale.x,
0));
}

/// <summary>
/// Faces towards the input
/// </summary>
void PlayerDirection()
{
    if (moveX < 0.0f && facingRight)
    {
        FlipPlayer();
    }
    else if (moveX > 0.0f && !facingRight)
    {
        FlipPlayer();
    }
}

/// <summary>
/// Changes the x-orientation of the player
/// </summary>
void FlipPlayer()
{
    facingRight = !facingRight;
    Vector2 localScale = gameObject.transform.localScale;

```



```

        localScale.x *= -1;
        transform.localScale = localScale;
        GetComponentInChildren<MeshRenderer>().transform.localScale = localScale;
    }

    /// <summary>
    /// Returns true if the player's feet overlap with the ground layer
    /// </summary>
    /// <returns></returns>
    bool OnGround()
    {
        Vector3 checkPos = new Vector3(transform.position.x, transform.position.y -
PlayerCC.bounds.extents.y, 0);
        return Physics2D.OverlapCircle(checkPos, checkradius, groundLayer);
    }

    void PlayerWalkAnimation()
    {
        if (moveX != 0)
        {
            animator.Play("PlayerWalk");
        }
        else animator.Play("PlayerIdle");
    }

    /// <summary>
    /// Checks if the player has transitioned from a falling state to a landed state and check the
    distance fell
    /// </summary>
    void FixedUpdate()
    {
        OnTheGround = OnGround();
        if (!wasFalling && isFalling) startofFall = transform.position.y;
        if (!wasGrounded && OnTheGround) GetComponent<HealthManager>().FallDamage(startofFall -
transform.position.y);
        wasGrounded = OnTheGround;
        wasFalling = isFalling;
    }
}

```

Runs On GameObject: LocalPlayer

Name: BlockInteractions.cs

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using TMPro;
using UnityEngine;

public class BlockInteractions : MonoBehaviour
{
    public GameObject MapManagerObject;
    [SerializeField]
    Vector2 mousePos;
    public int block;
    /// <summary>
    /// The Player inventory is held in an array, with the ordinal number representing the type of
    block and the value representing the quantity they have.
    /// </summary>
    public int[] inventory = new int[10];
    public int range;

    [SerializeField]
    public bool hasAxe { get; private set; }
    [SerializeField]
    public bool hasSword { get; private set; }

    [SerializeField]
    float timemining;
    Vector2 mousePostionOnCall;

    //[SerializeField] private string TempID;

    public string PlayerID { get; private set; }
    public TextMeshPro idText;

    void Start()
    {
        //transform.position =
        MapManagerObject.GetComponent<GenerationScriptV2>().PlayerSpawnPoint();
        //PlayerID = TempID;

        SetPlayerName();
        MapManagerObject = GameObject.Find("GeneratorV2");
    }

    public void SetPlayerName()
    {
        PlayerID = PassingVariables.playerID;
        idText.text = PlayerID;
    }

    // Update is called once per frame
    void Update()
    {
        if (!GetComponentInChildren<PlayerMenuManager>().hasAnyMenuOpen)
        {
            mousePos = GetComponent<Camera>().ScreenToWorldPoint(Input.mousePosition);
            //if (Input.GetMouseButton(0)) Break();

            if (Input.GetMouseButton(1) && block == 0 && MouseInRange()) TimedBreaking();
            if (Input.GetMouseButtonDown(1) && block != 0 && MouseInRange()) Build(block);
        }
    }
}

```

```

        //if (Input.GetKeyDown(KeyCode.P))
        //{
        //    LoadPlayerState("PlayerSaveFile.txt");
        //}
        //if (Input.GetKeyDown(KeyCode.O))
        //{
        //    SavePlayerState("PlayerSaveFile.txt");
        //}
    }

    /// <summary>
    /// Breaks a block in the Map Script and Increments the array at the given position
    /// </summary>
    void Break()
    {
        //Debug.Log(MapManagerObject.GetComponent<GenerationScriptV2>().ReturnTypeInfoBlock((int)System.Math
        //    .Truncate(mousePos.x), (int)System.Math.Truncate(mousePos.y)));

        inventory[MapManagerObject.GetComponent<GenerationScriptV2>().BreakBlock((int)System.Math.Truncate
        (mousePos.x), (int)System.Math.Truncate(mousePos.y))]]++;
    }

    /// <summary>
    /// Makes breaking a block take time
    /// </summary>
    void TimedBreaking()
    {
        if (timemining == 0) mousePostionOnCall = new
        Vector2((int)System.Math.Truncate(mousePos.x), (int)System.Math.Truncate(mousePos.y));
        if (mousePostionOnCall == new Vector2((int)System.Math.Truncate(mousePos.x),
        (int)System.Math.Truncate(mousePos.y)))
        {
            timemining += Time.deltaTime;
            //Debug.Log(timemining);
            if (timemining > AmountofTimetoBreak())
            {
                Break();
                //Debug.Log("Mined");
                timemining = 0;
            }
        }
        else timemining = 0;
    }

    /// <summary>
    /// Return the amount of time it take for each block type
    /// </summary>
    /// <returns></returns>
    float AmountofTimetoBreak()
    {
        float timetobreak;
        switch
        (MapManagerObject.GetComponent<GenerationScriptV2>().ReturnTypeInfoBlock((int)System.Math.Truncate(m
        ousePos.x), (int)System.Math.Truncate(mousePos.y)))
        {
            case (1):
                timetobreak = 0.6f;
                break;
            case (2):
                timetobreak = 1.8f;

```

```

        break;
    case (3):
        timetobreak = 1.8f;
        break;
    case (4):
        timetobreak = 1f;
        break;
    case (5):
        timetobreak = 0.2f;
        break;
    case (6):
        timetobreak = 1f;
        break;
    default:
        timetobreak = 1f;
        break;
    }
    if (hasAxe)
    {
        return timetobreak / 2;
    }
    else return timetobreak;
}

/// <summary>
/// Builds a block if its present in the inventory, deincreaments the inventory
/// </summary>
/// <param name="BlockPassed"></param>
void Build(int BlockPassed)
{
    if (inventory[BlockPassed] > 0 &&
MapManagerObject.GetComponent<GenerationScriptV2>().BuildBlock(BlockPassed,
(int)System.Math.Truncate(mousePos.x), (int)System.Math.Truncate(mousePos.y)))
    {
        inventory[BlockPassed]--;
    }
}

public void HudInput(int selectedBlock)
{
    block = selectedBlock;
}

public string QuantityinInventory()
{
    if (block == 0)
    {
        return "";
    }
    else return inventory[block].ToString();
}

/// <summary>
/// Used to check if the player has sufficient materials to craft each item
/// </summary>
/// <param name="item"></param>
/// <returns></returns>
public bool RequestToCraft(int item)

```

```

{
    if (item == 0 && !hasAxe)
    {
        if (inventory[2] >= 12 && inventory[3] >= 12 && inventory[6] >= 8)
        {
            inventory[2] -= 12;
            inventory[3] -= 12;
            inventory[6] -= 8;
            hasAxe = true;
            return true;
        }
    }
    if (item == 1 && !hasSword)
    {
        if (inventory[2] >= 12 && inventory[3] >= 10 && inventory[6] >= 4)
        {
            inventory[2] -= 12;
            inventory[3] -= 10;
            inventory[6] -= 4;
            hasSword = true;
            return true;
        }
    }
    if (item == 2)
    {
        if (inventory[4] >= 1)
        {
            inventory[4] -= 1;
            inventory[6] += 4;
            return true;
        }
    }
    return false;
}

/// <summary>
/// Checks if the distance from the player to the mouse is below a given value
/// </summary>
/// <returns></returns>
public bool MouseInRange()
{
    if (Mathf.Sqrt((float)(System.Math.Pow((mousePos.x - transform.position.x), 2) +
System.Math.Pow((mousePos.y - transform.position.y), 2))) < range) return true;
    else return false;
}

/// <summary>
/// Unused in final code, however has been left in incase of a version2
/// </summary>
/// <param name="worldName"></param>
public void SavePlayerState(string worldName)
{
    using (StreamWriter sw = new StreamWriter($"WorldSaves/{worldName}/{PlayerID}.txt"))
    {
        for (int i = 0; i < inventory.Length; i++)
        {
            sw.WriteLine(inventory[i]);
        }
        sw.WriteLine(transform.position.x);
        sw.WriteLine(transform.position.y);
        if (hasAxe) sw.WriteLine("1");
        else sw.WriteLine("0");
        if (hasSword) sw.WriteLine("1");
    }
}

```

```

        else sw.WriteLine("0");
    }
}

/// <summary>
/// Unused in final code, however has been left in incase of a version2
/// </summary>
/// <param name="worldName"></param>
public void LoadPlayerState(string worldName)
{
    string value;
    int i = 0;
    float tempx = 0;
    float tempy = 0;

    using (StreamReader sr = new StreamReader($"WorldSaves/{worldName}/{PlayerID}.txt"))
    {
        while ((value = sr.ReadLine()) != null)
        {
            //Debug.Log($"{i},{value},{(float)Convert.ToDouble(value)}");
            if (i < 10) inventory[i] = Convert.ToInt16(value);
            if (i == 10) tempx = (float)Convert.ToDouble(value);
            if (i == 11) tempy = (float)Convert.ToDouble(value);
            if (i == 12)
            {
                if (value == "1") hasAxe = true;
            }
            else hasAxe = false;
            if (i == 13)
            {
                if (value == "1") hasSword = true;
            }
            else hasSword = false;
            i++;
        }
        transform.position = new Vector2(tempx, tempy);
    }
}

/// <summary>
/// Used on player death
/// </summary>
public void EmptyInventory()
{
    hasAxe = false;
    hasSword = false;
    for (int i = 0; i < inventory.Length; i++)
    {
        inventory[i] = 0;
    }
}
}

```

Runs On GameObject: LocalPlayer

Name: HealthManager.cs

```

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

public class HealthManager : MonoBehaviour
{
    public Image healthBar;
    public int playerHealth = 10;
    public float regenTime;
    float timeWhenDamage;
    float healthToGain;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void FixedUpdate()
    {
        healthBar.rectTransform.localScale = new Vector3((float)playerHealth / 10, 1, 1);
        HealthRegeneration();
    }

    /// <summary>
    /// If the player takes damage from falling or from an attack it will call this function
    /// </summary>
    /// <param name="damage"></param>
    public void TakeDamage(int damage)
    {
        if (playerHealth - damage > 0)
        {
            playerHealth -= damage;
            timeWhenDamage = Time.time;
        }
        else PlayerDeath();
        FindObjectOfType<AudioManager>().Play("Take Damage");
    }

    /// <summary>
    /// Finds the amount of damage a fall will result in based on distance
    /// </summary>
    /// <param name="distance"></param>
    public void FallDamage(float distance)
    {
        // Debug.Log(Mathf.RoundToInt(distance / 4));
        if (distance > 6) TakeDamage(Mathf.RoundToInt(distance / 2));
    }

    /// <summary>
    /// Manages health regen
    /// </summary>
    void HealthRegeneration()
    {
        if (playerHealth < 10 && playerHealth != 0)
        {
            if (Time.time - timeWhenDamage > regenTime)
            {
                healthToGain += Time.deltaTime;
                playerHealth += Mathf.RoundToInt(healthToGain);
            }
        }
    }
}

```

```

    }
    else healthToGain = 0;
}

/// <summary>
/// Called if the health is 0
/// </summary>
void PlayerDeath()
{
    playerHealth = 10;
    FindObjectOfType<WorldEventManager>().PlayerDeath(gameObject);
}
}

```

Runs On GameObject: LocalPlayer

Name: AttackManager.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AttackManager : MonoBehaviour
{
    public GameObject Sword;
    public GameObject AttackPoint;
    public float AttackRate;
    public int AttackDamage;
    public LayerMask NPCLayer;

    Animator animator;
    float timeSinceAttack;

    // Start is called before the first frame update
    void Start()
    {
        animator = Sword.GetComponentInChildren<Animator>();
        Debug.DrawLine(Sword.transform.position, new Vector2(Sword.transform.position.x + 1,
Sword.transform.position.y + 2));
    }

    // Update is called once per frame
    /// <summary>
    /// Takes input from the users mouse and if they have crafted a sword then they attack
    /// </summary>
    void Update()
    {
        if (!GetComponentInChildren<PlayerMenuManager>().hasAnyMenuOpen)
        {
            timeSinceAttack += Time.deltaTime;
            if (Input.GetMouseButtonDown(0) && GetComponent<BlockInteractions>().hasSword &&
timeSinceAttack > AttackRate && GetComponent<BlockInteractions>().MouseInRange())
            {
                timeSinceAttack = 0;
                animator.Play("SwordSwing");
                FindObjectOfType<AudioManager>().Play("Sword Swing");
                Attack();
            }
        }
    }
}

```



```

    }

    /// <summary>
    ///
    /// </summary>
    private void FixedUpdate()
    {
        if (GetComponent<BlockInteractions>().hasSword)
        {
            Sword.GetComponent<SpriteRenderer>().enabled = true;
        }
        else Sword.GetComponent<SpriteRenderer>().enabled = false;
    }

    /// <summary>
    /// Adds all entities of type NPC layer, within the capsule range, and adds them to an array
    /// of colliders. It then calls the damage function for each
    /// </summary>
    void Attack()
    {
        Collider2D[] hitEnemies = (Physics2D.OverlapCapsuleAll(AttackPoint.transform.position, new
Vector2(1, 2), CapsuleDirection2D.Vertical, 0, NPCLayer));
        //Debug.Log(hitEnemies.Length);
        foreach(Collider2D hits in hitEnemies)
        {
            hits.GetComponent<ZombieControl>().TakeDamage();
            //hits.GetComponent<ZombieControl>().ApplyKnockback(transform.position);
        }
    }
}

```

Runs On GameObject: LocalPlayer

Name: UsernameText.cs

```

using System.Collections;
using System.Collections.Generic;
using TMPPro;
using UnityEngine;

public class usernameText : MonoBehaviour
{
    [SerializeField] private TMP_Text userName;

    public void ApplyUserName(string name) { userName.text = name; }
}

```

Runs On GameObject: LocalPlayer - HUD

Name: HUDManager.cs

```
//using Microsoft.Unity.VisualStudio.Editor;
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.UI;

public class HUDManager : MonoBehaviour
{
    public int scrollPosition = 1;
    [SerializeField]
    public GameObject Player;
    public UnityEngine.UI.Image Icon;
    public Sprite axeIcon, emptyIcon;
    public Sprite[] BlockIcons = new Sprite[6];
    public TextMeshProUGUI quantityText;
    int[] blockKey = { 0, 1, 2, 4, 5, 6 };

    public GameObject PauseDeathMenuManager;
    public GameObject craftMenu;
    bool craftMenuShowing;

    // Start is called before the first frame update
    void Start()
    {
        //Temporary Code
        //Player = GameObject.Find("Player");
    }

    // Update is called once per frame
    void Update()
    {
        if (!PauseDeathMenuManager.GetComponent<PlayerMenuManager>().hasAnyMenuOpen)
        {
            MouseScroll();
            SendtoPlayer();
            QuantityText();
        }

        //Debug.Log(block);
        //Debug.Log("running");
    }
    /// <summary>
    /// Changes the icon for the Axe tool if one has been crafted
    /// </summary>
    void FixedUpdate()
    {
        if (!PauseDeathMenuManager.GetComponent<PlayerMenuManager>().hasAnyMenuOpen)
        {
            if (PauseDeathMenuManager.GetComponent<PlayerMenuManager>().hasAnyMenuOpen) return;
            if (Player.GetComponent<BlockInteractions>().hasAxe) BlockIcons[0] = axeIcon;
            else BlockIcons[0] = emptyIcon;

            craftMenu.SetActive(craftMenuShowing);
        }
    }
}
```

```

/// <summary>
/// Controls which icon appears in the hud
/// </summary>
void MouseScroll()
{
    if (Mathf.RoundToInt(Input.GetAxisRaw("Mouse ScrollWheel") * 10) == 1) HudUP();
    if (Mathf.RoundToInt(Input.GetAxisRaw("Mouse ScrollWheel") * 10) == -1) HudDown();
}

void SendtoPlayer()
{
    Player.GetComponent<BlockInteractions>().HudInput(blockKey[Mathf.Abs(scrollPosition)]);
    Icon.sprite = BlockIcons[Mathf.Abs(scrollPosition)];
}

public void HudUP()
{
    //Debug.Log("Up Called");
    scrollPosition += 1;
    scrollPosition = (scrollPosition % blockKey.Length);
    FindObjectOfType<AudioManager>().Play("Hud Interact");
}

public void HudDown()
{
    //Debug.Log("Down Called");
    scrollPosition += -1;
    scrollPosition = (scrollPosition % blockKey.Length);
    FindObjectOfType<AudioManager>().Play("Hud Interact");
}

/// <summary>
/// Returns a sprite to the itemInHand Script passing the item currently selected in the HUD
/// </summary>
/// <returns></returns>
public Sprite PasstoHand()
{
    return BlockIcons[Mathf.Abs(scrollPosition)];
}

/// <summary>
/// Shows the Quantity of block in in the inventory
/// </summary>
void QuantityText()
{
    quantityText.text = (Player.GetComponent<BlockInteractions>().QuantityinInventory());
}

/// <summary>
/// Toggles the crafting screen
/// </summary>
public void ToggleCraftMenu()
{
    craftMenuShowing = !craftMenuShowing;
    FindObjectOfType<AudioManager>().Play("Hud Interact");
}

public void CraftRequestAxe()
{
    if (Player.GetComponent<BlockInteractions>().RequestToCraft(0))
        FindObjectOfType<AudioManager>().Play("Craft Success");
}

```

```

        else FindObjectOfType<AudioManager>().Play("Craft Fail");
    }

    public void CraftRequestSword()
    {
        if (Player.GetComponent<BlockInteractions>().RequestToCraft(1))
            FindObjectOfType<AudioManager>().Play("Craft Success");
        else FindObjectOfType<AudioManager>().Play("Craft Fail");
    }

    public void CraftRequestPlank()
    {
        if (Player.GetComponent<BlockInteractions>().RequestToCraft(2))
            FindObjectOfType<AudioManager>().Play("Craft Success");
        else FindObjectOfType<AudioManager>().Play("Craft Fail");
    }
}

```

Runs On GameObject: LocalPlayer - ItemInHand

Name: ItemInHandScript.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class iteminhandscript : MonoBehaviour
{
    // Start is called before the first frame update
    SpriteRenderer spriteRenderer;

    void Start()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        spriteRenderer.sprite = FindObjectOfType<HUDManager>().PasstoHand();
    }
}

```

Runs On GameObject: LocalPlayer - HUD

Name: PlayerMenuManager.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerMenuManager : MonoBehaviour
{
    public GameObject PauseMenu;
    public GameObject DeathMenu;
    public GameObject Player;
    public bool hasAnyMenuOpen { get; private set; }
}

```

```

bool hasDeathMenuOpen;
bool hasPauseMenuOpen;

// Start is called before the first frame update
void Start()
{
    hasDeathMenuOpen = false;
    hasPauseMenuOpen = false;
    hasAnyMenuOpen = false;
}

// Update is called once per frame
void Update()
{
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        TogglePauseMenu();
    }
}

/// <summary>
/// Toggles the Pause menu on Esc.
/// </summary>
void TogglePauseMenu()
{
    Sound();
    if (!hasDeathMenuOpen)
    {
        if (hasPauseMenuOpen)
        {
            PauseMenu.SetActive(false);
            hasAnyMenuOpen = false;
            hasPauseMenuOpen = false;
        }
        else
        {
            PauseMenu.SetActive(true);
            hasAnyMenuOpen = true;
            hasPauseMenuOpen = true;
        }
    }
}

/// <summary>
/// Accessed by the Resume button
/// </summary>
public void Resume()
{
    PauseMenu.SetActive(false);
    hasPauseMenuOpen = false;
    hasAnyMenuOpen = false;
    Sound();
}

// <summary>
// Accessed by the Leave button
// </summary>
public void Leave()
{

```

```

//Temporary Before Network Code
Debug.Log("Leave Called");

PauseMenu.SetActive(false);
hasPauseMenuOpen = false;
hasAnyMenuOpen = false;
//Save();
FindObjectOfType<NetworkManager>().CalledLeave();
Sound();
}

/// <summary>
/// Enable the overlay upon death
/// </summary>
public void EnableDeathScreen()
{
    DeathMenu.SetActive(true);
    hasAnyMenuOpen = true;
    hasDeathMenuOpen = true;
}

/// <summary>
/// Respawns the player
/// </summary>
public void Respawn()
{
    DeathMenu.SetActive(false);
    hasAnyMenuOpen = false;
    hasDeathMenuOpen = false;
    FindObjectOfType<WorldEventManager>().PlayerRespawn(Player);
    Sound();
}

/// <summary>
/// Unused
/// </summary>
public void Save()
{
    FindObjectOfType<WorldEventManager>().SaveAll();
    Sound();
}

void Sound() { FindObjectOfType<AudioManager>().Play("Hud Interact"); }
}

```

Runs On GameObject: LocalPlayer - HUD

Name: PlayerTextChat.cs

```

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

public class PlayerTextChat : MonoBehaviour
{
    // Start is called before the first frame update
    public TMP_InputField input;
}

```

```

public void Send()
{
    FindObjectOfType<TextChatManger>().AddToChat(input.text);
    FindObjectOfType<AudioManager>().Play("Hud Interact");
    //GetComponent<Player>().UpdateTextChat(input.text);
    GetComponentInParent<Player>().UpdateTextChat(input.text);
}

}

```

Runs On GameObject: TextChatHUD

Name: TextChatManager.cs

```

using System.Collections;
using System.Collections.Generic;
using TMPro;

using UnityEngine;
using UnityEngine.Rendering.Universal;
//using static UnityEditor.Progress;

public class TextChatManger : MonoBehaviour
{
    // Start is called before the first frame update
    public int chatLength;
    public Queue<string> messages;
    public TMP_Text chat;

    void Start()
    {
        messages = new Queue<string>(chatLength);
        //chat = GetComponent<TextMeshPro>();
        UpdateChat();
    }

    // Update is called once per frame
    void UpdateChat()
    {
        chat.text = "";
        int count = 1;
        foreach (var item in messages)
        {
            chat.text += "\n" + count.ToString() + " : " + item;
            count++;
        }
    }

    /// <summary>
    /// Adds to the chat, with overflow protection
    /// </summary>
    /// <param name="Message"></param>
    public void AddToChat(string Message)
    {
        if (messages.Count == chatLength)
        {
            messages.Dequeue();
            messages.Enqueue(Message);
        }
    }
}

```

```

        else messages.Enqueue(Message);

        UpdateChat();
    }
}

```

Runs On GameObject: WorldEventManager

Name: WorldEventManager.cs

```

using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;

public class WorldEventManager : MonoBehaviour
{
    public bool EnableDayNightCycle = true;
    public GameObject PlayerPrefab;
    //public GameObject MapManager;

    // Start is called before the first frame update
    //void Start()
    //{
        //    GetComponentInChildren<GenerationScriptV2>().SetWorldName(PassingVariables.worldName);
        //    Debug.Log(PassingVariables.worldName);
        //    GetComponentInChildren<GenerationScriptV2>().Generation();
        //    Instantiate(PlayerPrefab,
GetComponentInChildren<GenerationScriptV2>().PlayerSpawnPoint(), Quaternion.identity);
        //    //Debug.Log("finito");

        //}

    // Update is called once per frame
    void FixedUpdate()
    {
        //GetComponentInChildren<DayNightCycle>().DayNightEnabled = EnableDayNightCycle;
    }

    /// <summary>
    /// Moves the player away from the world on death, then empties the inventory
    /// </summary>
    /// <param name="Player"></param>
    public void PlayerDeath(GameObject Player)
    {
        Player.GetComponentInChildren<PlayerMenuManager>().EnableDeathScreen();
        Player.GetComponent<BlockInteractions>().EmptyInventory();
        Player.transform.position = new Vector3(-1000, 0, 0);
    }

    /// <summary>
    /// Teleports the player from limbo, back to the centre of the map
    /// </summary>
    /// <param name="Player"></param>
    public void PlayerRespawn(GameObject Player)

```



```

{
    Player.transform.position =
GetComponentInChildren<GenerationScriptV2>().PlayerSpawnPoint();
}

/// <summary>
/// Unused
/// </summary>
public void SaveAll()
{
    GetComponentInChildren<GenerationScriptV2>().SaveMap();

    //Finding All the Players in the scene
    GameObject[] players;
    players = GameObject.FindGameObjectsWithTag("Player");

    foreach (var item in players)
    {
        item.GetComponent<BlockInteractions>().SavePlayerState(GetComponentInChildren<GenerationScriptV2>()
        ).CurrentWorldName());
    }
}
}

```

Runs On GameObject: Zombie

Name: ZombieControl.cs

```

using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;

public class ZombieControl : MonoBehaviour
{
    public int sightRange;

    public LayerMask playerLayer;
    public LayerMask groundLayer;

    public GameObject attackPoint;
    public float attackDamage;
    public float attackRate;

    public float knockbackStrength;
    public float knockbackDuration;
    public float walkSpeed;
    public float jumpHeight = 6;
    float jumpForce;

    int ZombieHealth = 10;
    public bool OnTheGround;

    Rigidbody2D zombieRB;
    Animator animator;
    DayNightCycle DayNightCycleInUse;
    GameObject DayNightManager;

    float patrolRange;
    float timeSinceActive;

```

```

float time;

bool hasBeenKnockedBack;
float timeSinceKB;
float timeSinceAttack;

public bool playerInRange { get { return (Physics2D.OverlapCircle(transform.position,
sightRange, playerLayer)); } }
public bool OnGround { get { return Physics2D.OverlapCircle(new Vector2(transform.position.x,
transform.position.y - 1.1f), 0.1f, groundLayer); } }
public bool playerInAttackRange { get { return
Physics2D.OverlapCapsule(attackPoint.transform.position, new Vector2(1, 2),
CapsuleDirection2D.Vertical, 0, playerLayer); } }

/// <summary>
/// Assigns variables and ensure that zombies don't run into each other
/// </summary>
private void Start()
{
    zombieRB = GetComponent<Rigidbody2D>();
    animator = GetComponent<Animator>();
    Physics.IgnoreLayerCollision(gameObject.layer, gameObject.layer);
}

/// <summary>
/// Controls animations
/// </summary>
void ZombieWalkAnimations()
{
    if (zombieRB.velocity.x != 0)
    {
        animator.Play("ZombieWalk");
    }
    else animator.Play("ZombieIdle");
}

/// <summary>
/// Faces the zombie towards its waiking direction, using local scale
/// </summary>
void FaceTowardsWalkingDirection()
{
    if (zombieRB.velocity.x > 0)
    {
        transform.localScale = new Vector3(1, 1, 1);
    }
    if (zombieRB.velocity.x < 0)
    {
        transform.localScale = new Vector3(-1, 1, 1);
    }
}

/// <summary>
/// As movement is controlled by the server, this only control what the zombie attack.
/// </summary>
private void FixedUpdate()
{
    OnTheGround = OnGround;
    Debug.DrawRay(new Vector2(transform.position.x, transform.position.y - 0.5f), new
Vector2(transform.localScale.x, 0), Color.red);
    Debug.DrawRay(new Vector2(transform.position.x, transform.position.y + 0.5f), new
Vector2(transform.localScale.x, 0), Color.red);
    ZombieWalkAnimations();
    FaceTowardsWalkingDirection();
}

```

```

    if (playerInRange)
    {
        FaceTowardsPlayer(playerInRangePosition());
        if (playerInAttackRange) AttackManager();
        //if (!playerInAttackRange) zombieRB.velocity = new Vector2(walkSpeed *
transform.localScale.x, zombieRB.velocity.y);
    }

}

void FaceTowardsPlayer(Vector2 playerPos)
{
    if (transform.position.x < playerPos.x) transform.localScale = new Vector3(1, 1, 1);
    if (transform.position.x > playerPos.x) transform.localScale = new Vector3(-1, 1, 1);
    Debug.DrawLine(transform.position, playerPos, Color.red);
}

/// <summary>
/// Returns the closes player's position as a Vector3
/// </summary>
/// <returns></returns>
public Vector2 playerInRangePosition()
{
    Vector2 closePlayer = new Vector2(0, 0);
    float previousDistance = sightRange;
    if (playerInRange)
    {
        Collider2D[] players = (Physics2D.OverlapCircleAll(transform.position, sightRange,
playerLayer));
        float currentDistance;
        foreach (var item in players)
        {
            currentDistance = (Mathf.Sqrt((float)(System.Math.Pow((item.transform.position.x -
transform.position.x), 2) + System.Math.Pow((item.transform.position.y - transform.position.y),
2))));
            if (currentDistance < previousDistance)
            {
                closePlayer = item.transform.position;
                previousDistance = currentDistance;
            }
        }
    }
    return closePlayer;
}

/// <summary>
/// Sends an attack command, catches errors
/// </summary>
void Attack()
{
    Collider2D[] hitPlayers = (Physics2D.OverlapCapsuleAll(attackPoint.transform.position, new
Vector2(1, 2), CapsuleDirection2D.Vertical, 0, playerLayer));
    //Debug.Log(hitEnemies.Length);
    foreach (Collider2D hits in hitPlayers)
    {
        try
        {
            hits.GetComponent<HealthManager>().TakeDamage((int)attackDamage);
        }
        catch
        {
        }
    }
}

```

```

    }

}

/// <summary>
/// Attack at the given attack rate
/// </summary>
void AttackManager()
{
    timeSinceAttack += Time.deltaTime;
    if (timeSinceAttack > attackRate)
    {
        timeSinceAttack = 0;
        Attack();
    }
}

/// <summary>
/// Despawns the zombie on being attack
/// </summary>
public void TakeDamage()
{
    GetComponent<Zombie>().DespawnFromClient();
}
}

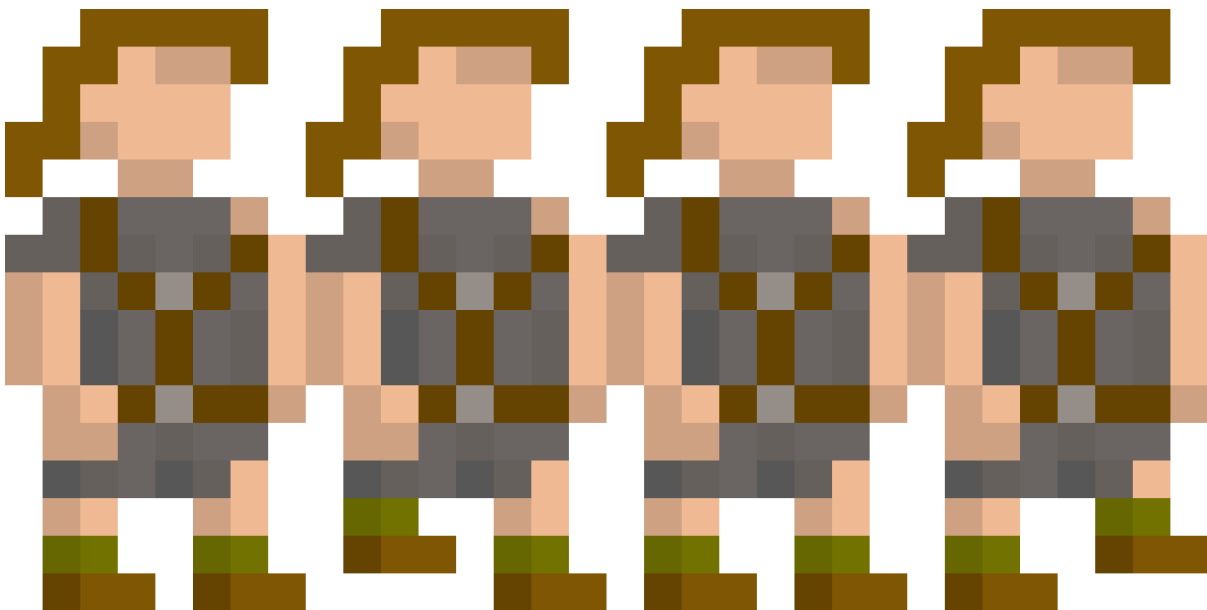
```

Assets

As listed in Objective 6.1 and 6.2, the art style of the project is important to my client, Elliot has noted that the art style should be consistent throughout, taking inspiration from games of the late 80s and early 90s, in addition to the newer wave of retro looking games.

Here's the final texture for each asset in the game, as mentioned in **Asset Requirements** all assets are produced by myself.

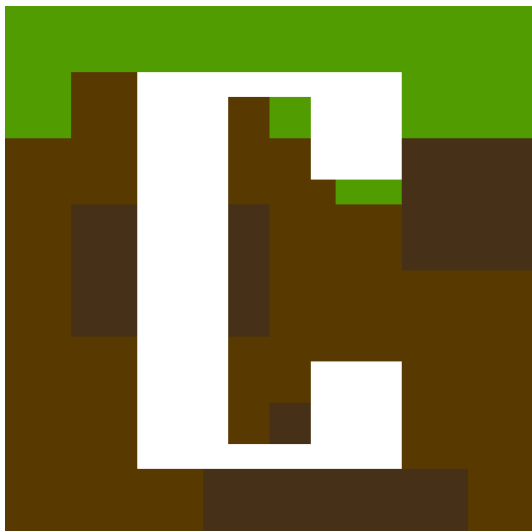
Character.png



BlockTextures.png



ClientIcon.png



ServerIcon.png

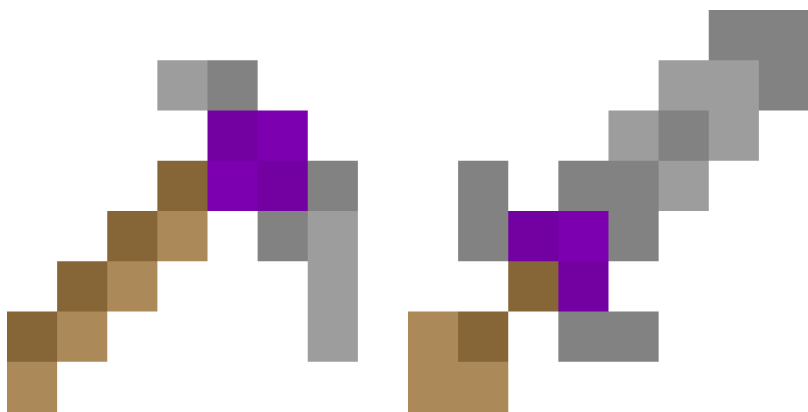


InventoryTexture.png

Note that each piece of the texture can be cut out, as shown then top two pieces are the button icons. The bottom left cluster is the player HUD, along with the crafting menu in the bottom right.

SwingAnimation.png

toolTextures.png



ZombieMob.png



Font Used – From DaFont.Com
<https://www.dafont.com/pixellari.font>

Pixellari.ttf

A 0065	B 0066	C 0067	D 0068	E 0069	F 0070	G 0071	H 0072	I 0073	J 0074	K 0075	L 0076	M 0077
N 0078	O 0079	P 0080	Q 0081	R 0082	S 0083	T 0084	U 0085	V 0086	W 0087	X 0088	Y 0089	
Z 0090												

a 0097	b 0098	c 0099	d 0100	e 0101	f 0102	g 0103	h 0104	i 0105	j 0106	k 0107	l 0108	m 0109	n 0110	o 0111
p 0112	q 0113	r 0114	s 0115	t 0116	u 0117	v 0118	w 0119	x 0120	y 0121	z 0122				

0 0048	1 0049	2 0050	3 0051	4 0052	5 0053	6 0054	7 0055	8 0056	9 0057
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

Technique Table

Technique Group	Coding Technique	Location in Code
A	Queue	TextChatManager.cs
A	Dynamic generation of objects based on complex user-defined use of OOP model	Zombie.cs Player.cs
A	Server-side scripting using request and response objects	NetworkManager.cs GameLogic.cs

	and server-side extensions for a complex client-server model	Player.cs Zombie.cs
A	Complex User Defined algorithm	GenerationScriptV2.cs (cellular automata algorithm)
A	Advanced matrix operations	Use of Vector2 and Vector3 throughout ZombieControl.cs playerInRangePosition
B	Writing and Reading from files	GenerationScriptV2.cs
B	Generation of objects based on a simple OOP model	Throughout...
B	Simple User Defined Algorithms	Throughout...
B	Multi-Dimensional Arrays	GeneratioScriptV2.cs
B	Dictionary	Player.cs Zombie.cs
C	Single-Dimensional Arrays	Throughout...
Undefined	Use of Enumerators	NetworkManager.cs
Undefined	Procedural Map Generation	GeneratioScriptV2.cs

Testing

Testing Table

To test my programs robustness, I will have to run each application to test its' functions and data:

Note that any test numbered with a V, will have its' testing evidence covered in this video:

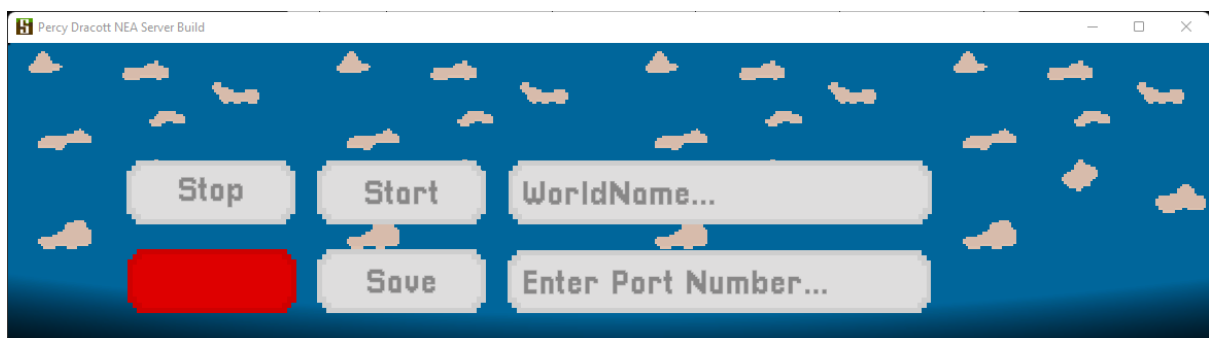
<https://youtu.be/Tz43xxyZaC4>

Server Application

Test No.	Test Purpose	Test Type	Test Data (if applicable)	Expected Outcome
1.1	Loads to Menu	Functionality	N/A	Loads to main menu
1.2	Main menu graphics inspection	Design	N/A	The Main menu displays all starting graphics
1.3	Main Menu input fields are unlocked	Functionality	N/A	Input fields allow an input to be typed
1.4	Start Button Check	Functionality	Valid Port Number Input.	Server should start, and indicator light should go green. A map is generated
1.5	Erroneous Input Check	Erroneous Input	Empty Port Number Input.	No Change.

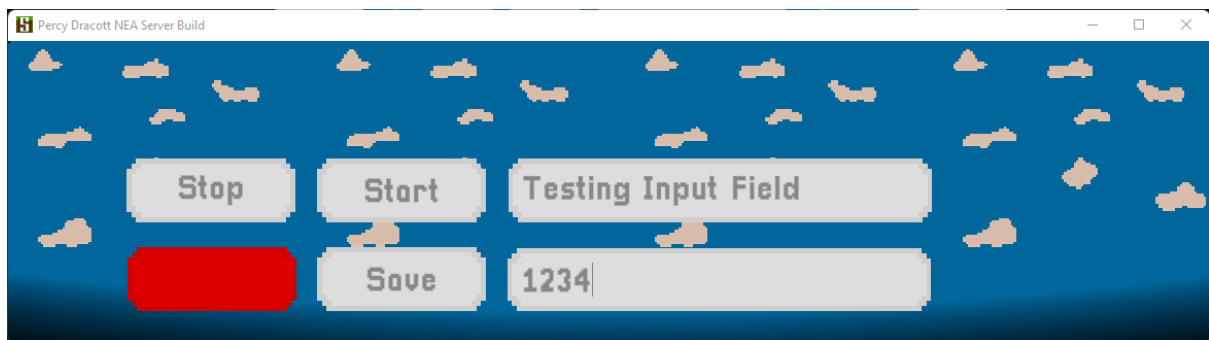
1.6	Erroneous Input Check	Erroneous Input	Non-number input into Port Input Field	Input should not be entered into the Port Input Field.
1.7	Start Button Check	Functionality	New World Name Input, Valid Port Number Input.	Server starts, indicated by the green indicator. A new map is generated and saved to a new location.
1.8	Start Button Check	Functionality	Already used and saved World Name Input, Valid Port Number Input.	Server starts, indicated by the green indicator. A map is loaded from the save, including all previous builds.
1.9	Stop Button Check	Functionality	N/A	Server stops, indicated by the red indicator icon.
V.1	Client Joining Check	Functionality	Client Username	Client Spawns a player into the scene, given their specified username.
V.2	Client Joining Check	Erroneous	Empty Client Username	Client Spawns a player into the scene, given "Guest" as a username
V.3	Client Leave Check	Functionality	N/A	Client's player is removed from the scene

Server Testing Evidence

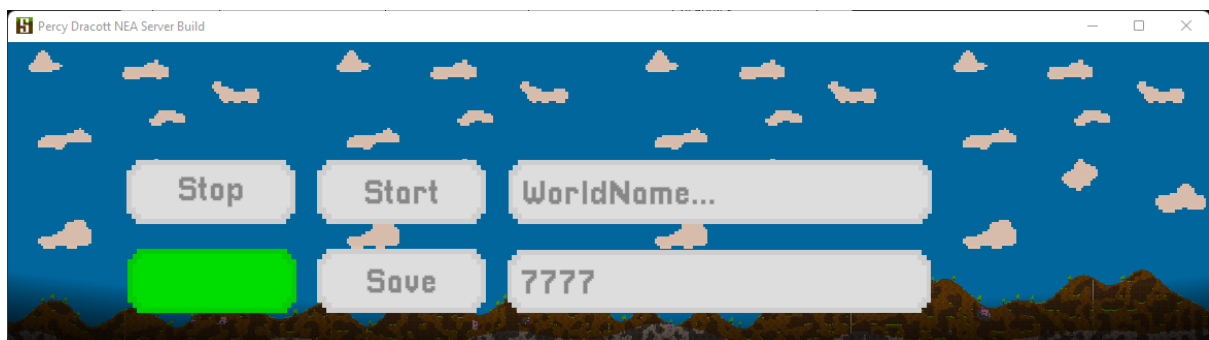


Test 1.1: Server Application loads to main menu with no issues, expected outcome. Test Successful

Test 1.2: The Main menu starting graphics all load in the correct positions, expected outcome. Test Successful



Test 1.3: Menu Input fields are unlocked on loading the application, expected outcome. Test Successful.

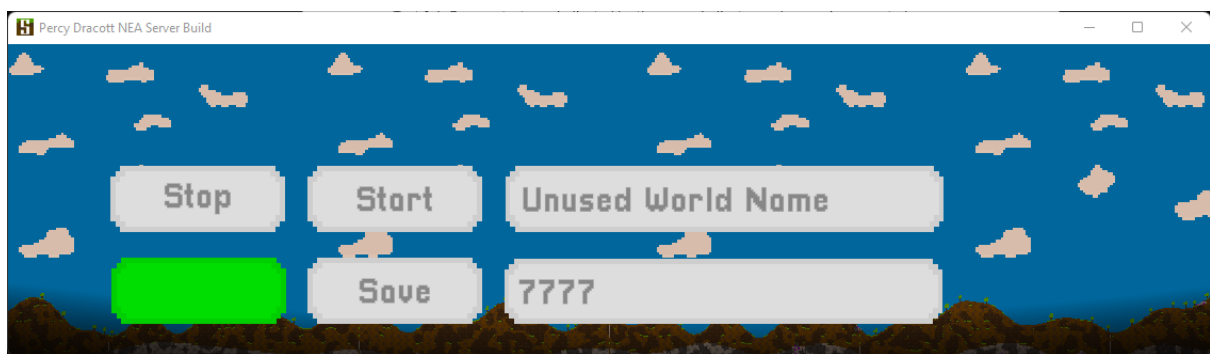


Test 1.4: Server starts, as indicated by the green indicator and a map is generated, expected outcome. Test Successful

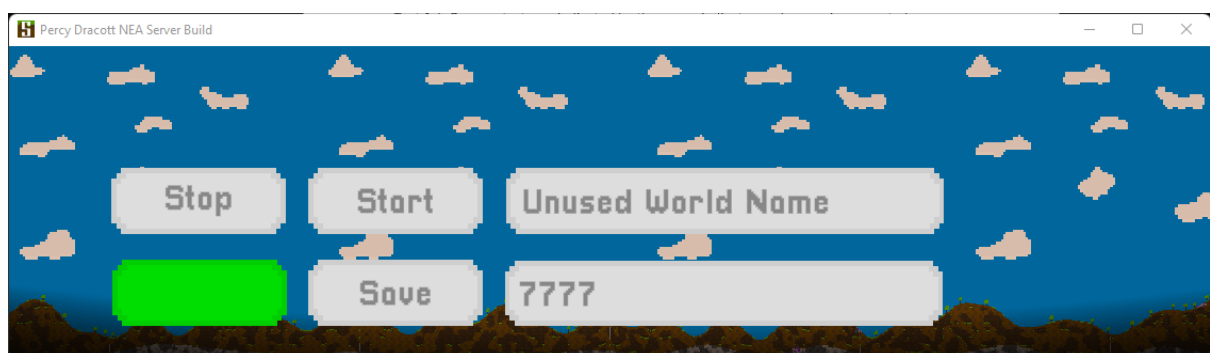


Test 1.5: Server doesn't start, no change, expect outcome. Test Successful

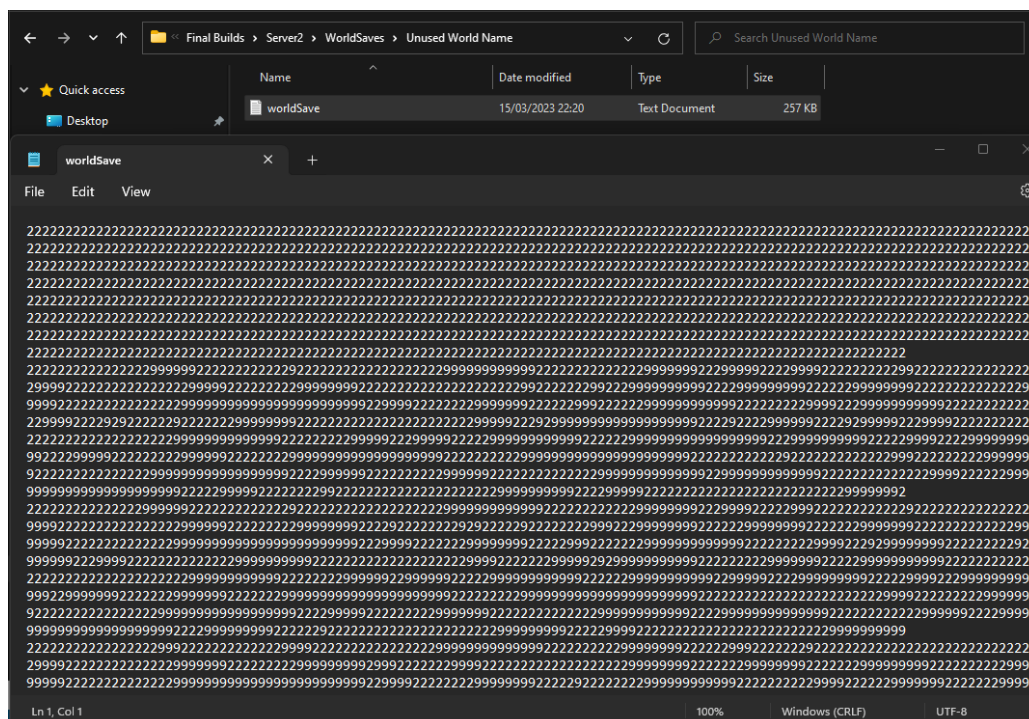
Test 1.6: Port Input field only accepts number characters, expected outcome. Test Successful



Test 1.7: Server Starts and a newly generated world is created, expected outcome. Test Successful



Note that the program was closed and reopened, with the same world name entered.



Test 1.8: Server starts, and the world is loaded from the save file, expected outcome. Test Successful



Test 1.9: Server stops, as indicated by the red icon, expected outcome. Test Successful

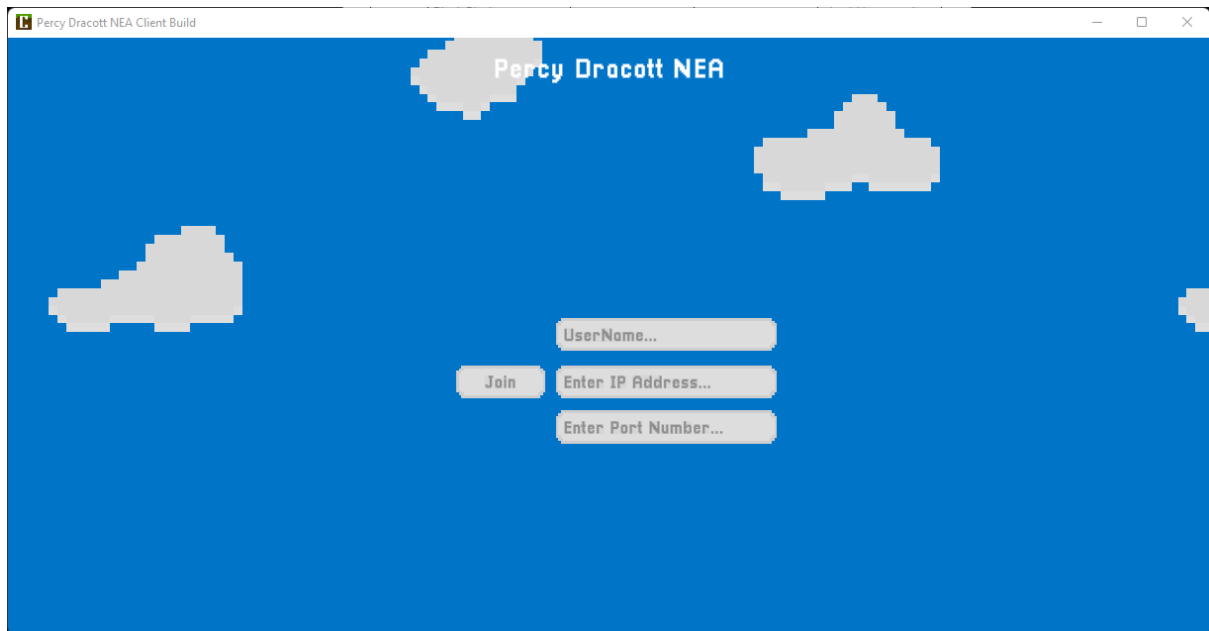
Client Application

Test No.	Test Purpose	Test Type	Test Data (if applicable)	Expected Outcome
1.1	Loads to Menu	Functionality	N/A	Loads to main menu
1.2	Main menu graphics inspection	Design	N/A	The Main menu displays all starting graphics
1.3	Main Menu input fields are unlocked	Functionality	N/A	Input fields allow an input to be typed
1.4	Connect Button Check	Functionality	Valid Port, IP, and Username. (Active Server Running)	Player should connect and be spawned into the world
1.5	Connection Check	Erroneous	Invalid Port, Valid IP, and Username. (Active Server Running)	Client should attempt to connect, however upon being unable, should return to the main menu, with a chat notification.
1.6	Connection Check	Erroneous	Invalid IP, Valid Port, and Username. (Active Server Running)	Client should attempt to connect, however upon being unable, should return to the main menu, with a chat notification.
1.7	Attempting to connect to a full server	Boundary	N/A	Client should attempt to connect, however upon being unable, should return to the main menu, with a chat notification.
2.1	Client Pause Menu	Functionality	"Esc" keyboard input	Pause menu should appear.
2.2	Client Pause Menu	Functionality	"Esc" keyboard input	Pause menu should toggle off.

2.3	Client Pause Menu, Resume Button	Functionality	N/A	Client's pause menu should disappear, client should remain in the game
2.4	Client Pause Menu, Leave Button	Functionality	N/A	Client's pause menu should disappear, client should leave the game and should be returned to main menu.
V.4	Client HUD Checks	Functionality	Mouse Scroll Wheel	Item in the players hand and HUD icon should change.
V.5	Client HUD Checks	Functionality	Button Click	Item in the players hand and HUD icon should change.
V.6	Crafting Button Check	Functionality	Button Click	Crafting menu should toggle on/off
V.7	Crafting Check	Functionality	Button Click	Client should obtain crafted item; quantities of recourses decrease as per the crafting requirements.
V.8	Crafting Check, with insufficient items	Boundary	Button Click	Client should be unable to craft item, no recourses are removed. Audio queue is given.
V.9	Client placing blocks	Functionality	Right Click	Block should appear in the map, clients inventory quantity for the given block should decrease.
V.10	Client breaking blocks, without tool	Functionality	Right Click	Block should disappear from the map, clients inventory quantity for the given block should increase. Should take a reasonable amount of time to break.
V.11	Client breaking blocks, with tool	Functionality	Right Click	Block should disappear from the map, clients inventory quantity for the given block should increase. Time taken to break the block is halved.

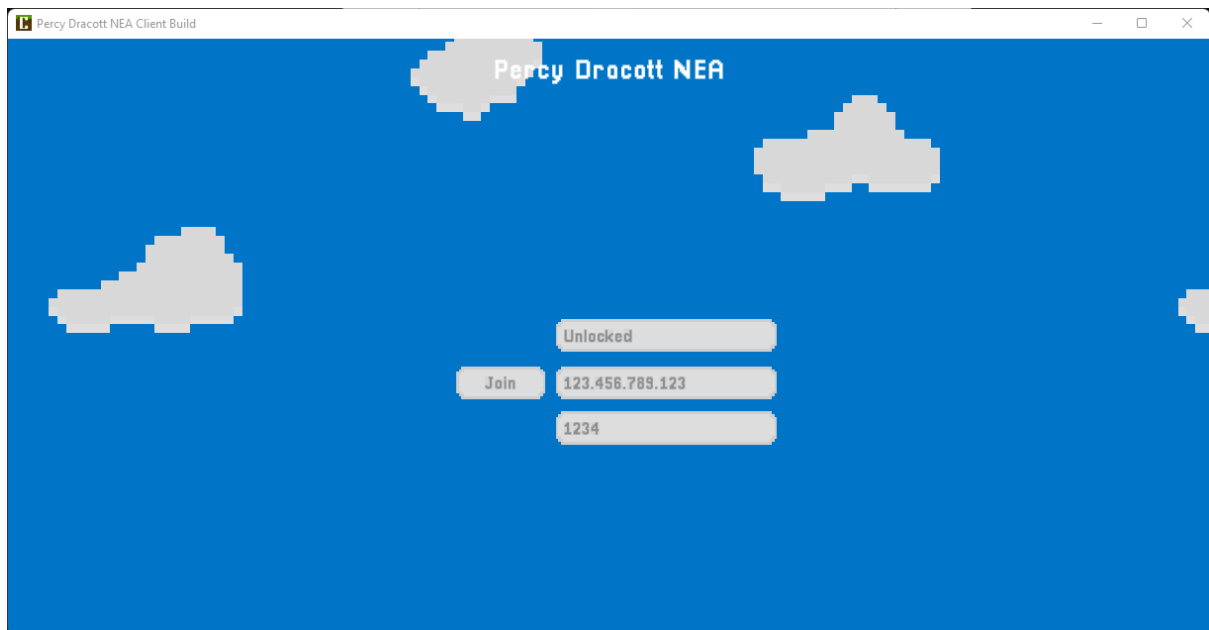
V.12	Player Taking Damage From falling	Functionality	N/A	Player's health should decrease
V.13	Player Death	Functionality	N/A	Player should be presented by a Death Screen
V.14	Death Screen Respawn Button	Functionality	Button Click	Player should be spawning back into the world with an empty inventory.
V.15	Death Screen Leave Button	Functionality	Button Click	Player should be disconnected from the server and returned to the main menu.
V.16	Connecting Multiple Clients	Functionality	N/A	Two players should appear in the scene.
V.17	Syncing Multiple Clients Movements	Functionality	N/A	Each players movements should be synced across both connected clients
V.18	Syncing Multiple Clients Block Placing	Functionality	N/A	Each players maps should be synced across both connected clients
V.19	Syncing Multiple Clients Block Breaking	Functionality	N/A	Each players maps should be synced across both connected clients
V.20	One of Multiple Clients Disconnecting	Functionality	N/A	Player should be removed from the scene; remaining player should remain connected.
V.21	Zombie Spawning across multiple clients	Functionality	N/A	Zombies position should be synced across all connected clients.
V.22	Zombie Despawning across multiple clients	Functionality	N/A	Zombies despawning should be synced across all connected clients.
V.23	Text Chat	Functionality	Text Input	Text Chat should sync across all clients.

Client Testing Evidence

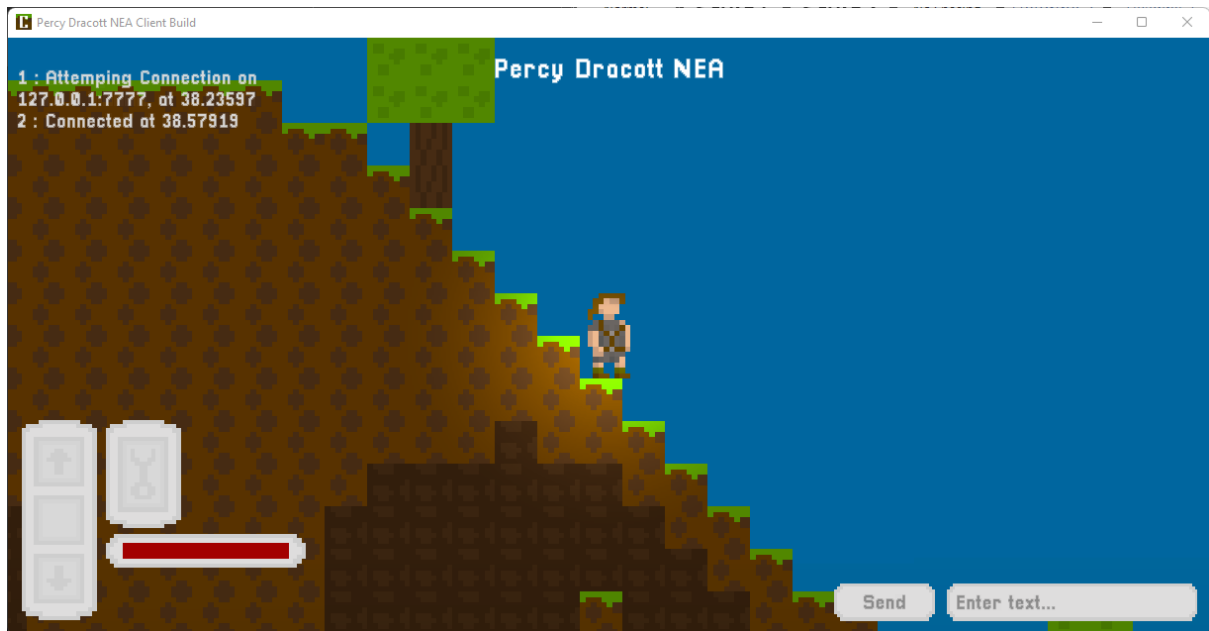


Test 1.1: Application loads to main menu, expected outcome. Test Successful

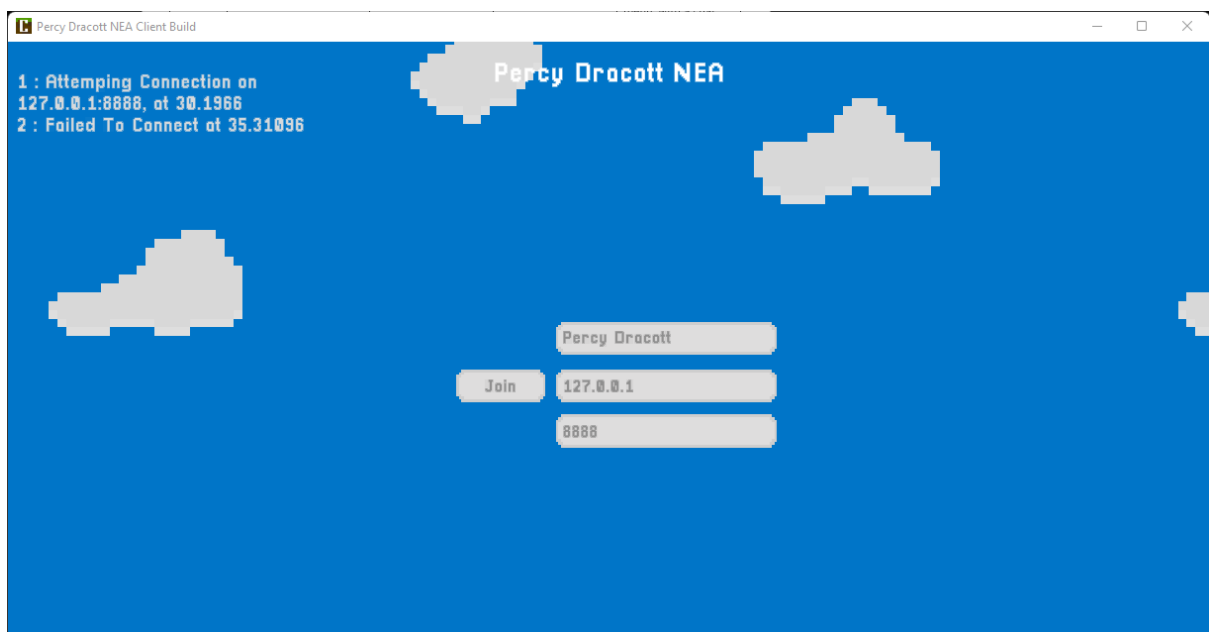
Test 1.2: All menu graphics are loaded in, expected outcome. Test Successful



Test 1.3: Input fields are unlocked on loading the application, expected outcome. Test Successful



Test 1.4: Player connected and be spawned into the world, expected outcome. Test Successful



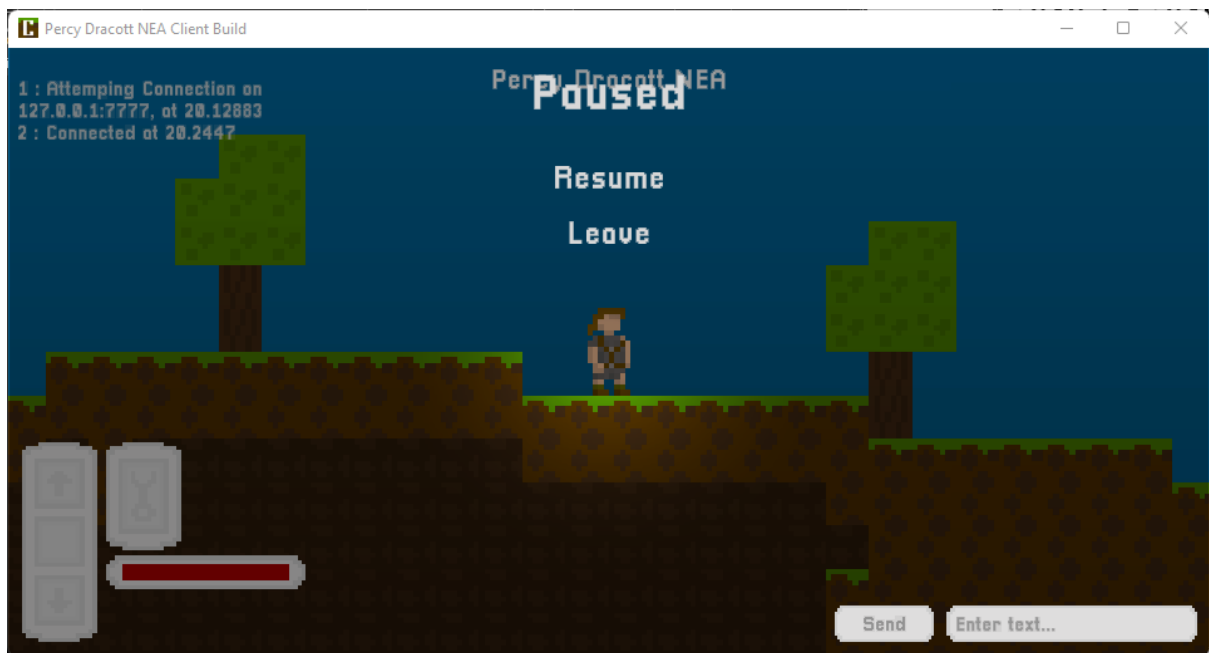
Test 1.5: (Server running on local machine, with port 7777 used) Attempted to connect, returned to main menu, message added to log, expected outcome. Test Successful



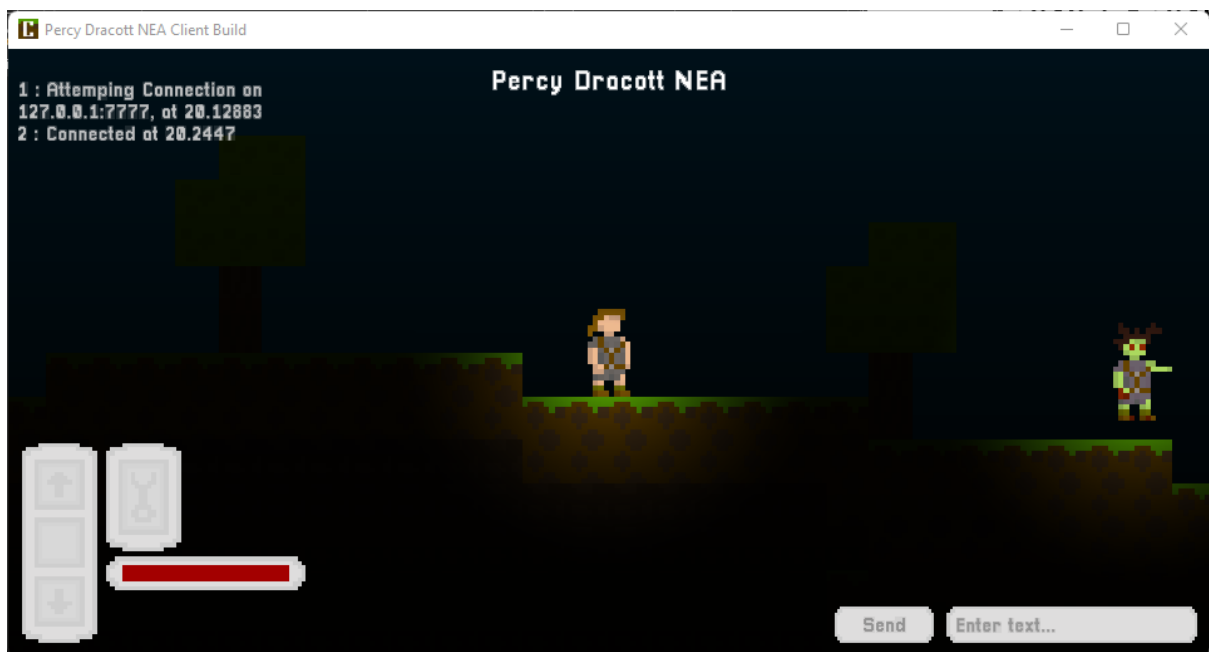
Test 1.6: (Server running on local machine with local IP 127.0.0.1, with port 7777 used)
Attempted to connect, returned to main menu, message added to log, expected outcome.
Test Successful



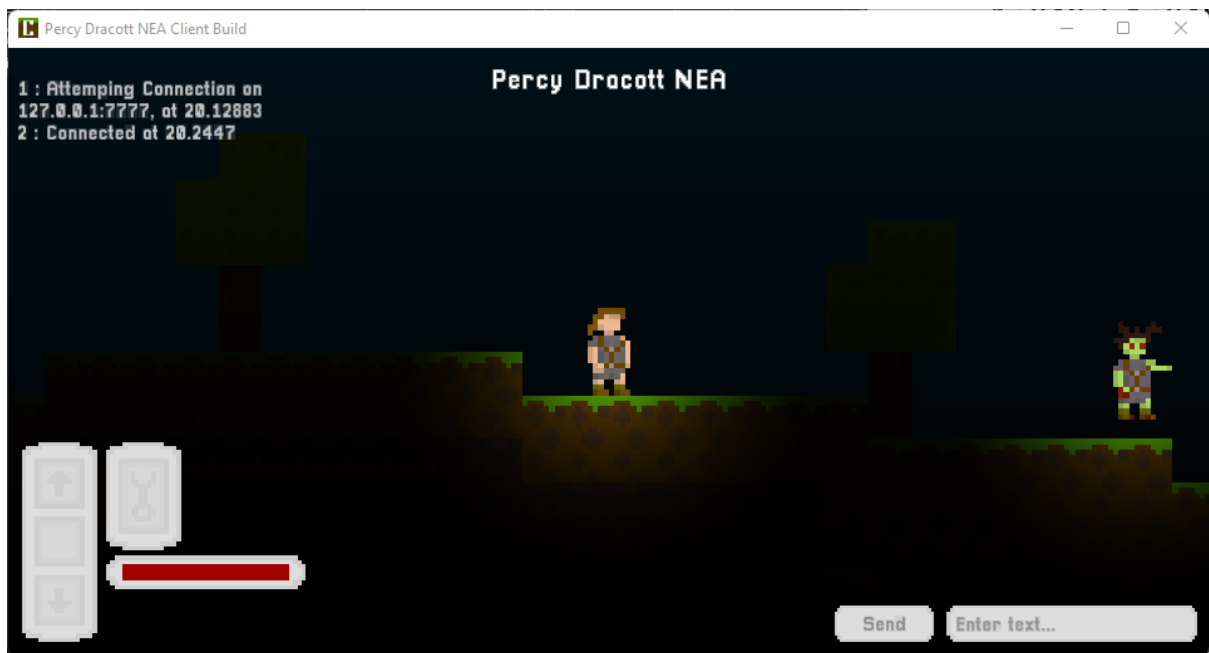
Test 1.7: Attempted to connect, returned to main menu, message added to log, expected outcome. Test Successful



Test 2.1: Pause menu appears on key press, expected outcome. Test Successful



Test 2.2: Pause menu is disabled on key press, expected outcome. Test Successful



Test 2.3: On "Resume" button press, pause menu is disabled, expected outcome. Test Successful



Test 2.4: On "Leave" button press, player disconnects and is returned to main menu, expected outcome. Test Successful

Evaluation

Solution Overview

As far as the problem and brief are concerned, I believe my solution fully fulfils its' role, being a multiplayer sandbox survival game. The two applications completely meet my client's needs as well as hitting all or their design preferences. The two applications allow my client to use the game as they wish, either alone running a client and server on the same computer; or share the experience with their friends using multiplayer.

The one part of this solution I believe is the most complex is the player and zombie classes, where they are both object-oriented classes which are stored in dictionaries. As these tie in with the multiplayer network code they require a different coding challenge, as a scalable number of entities will be running and executing the code as any one time.

Objective Evaluation

After completing my solution and testing it thoroughly, I can now evaluate the solution against the objectives given at the start of the project, which are as follows:

01. Have functioning game element, which:
 - 1.1. Allows the player to move around the world,
 - 1.2. Contains hostile Mobile-Entities (mobs) in the world,
 - 1.3. Has some form of combat between the player and the mobs,
 - 1.4. Has a unique procedurally generated terrain and cave system,
 - 1.5. Has a destructible world where blocks can be mined by the player,
 - 1.6. Has a building system where the player can build structures,
02. Have a randomly generating map, which:
 - 2.1. Is randomly generated at game start,
 - 2.2. Has a tunnelling cave generation algorithm,
 - 2.3. Has randomly generated trees,
03. Have a functioning Menu System, which:
 - 3.1. Allows for new games to be created,
 - 3.2. Allows for players to connect online,
 - 3.3. Displays the game title,
 - 3.4. Allows the user to close the program,
04. Have a functioning Multiplayer System, which:
 - 4.1. Allows multiple players to play in the same world,
 - 4.2. Allows the world to be synced across each player,
 - 4.3. Allows interactions between players,
 - 4.4. Allow players to connect through LAN or WAN
05. Satisfy the clients brief by:
 - 5.1. Have a save-able and load-able world,
 - 5.2. Have a Day / Night cycle,
 - 5.3. Include fantasy mobs such as skeletons, or zombies,
 - 5.4. Be playable on PC,
 - 5.5. Be playable on a mouse and keyboard or trackpad,
06. Be appealing to my End-Users by:

- 6.1. Having a consistent art style throughout,
- 6.2. Using a consistent pixel by pixel tile size throughout,

Objective No.	Comment
1.1	Player is able to move around the world using keyboard inputs
1.2	A zombie mob can attack the player, and is spawned randomly during the night
1.3	The player is able to attack mobs with a sword, once crafted
1.4	Terrain is procedurally generated using Perlin noise, caves are generated using a cellular automata.
1.5	Blocks can be mined by any player in the world
1.6	Blocks can be placed by any player
2.1	Map is mathematically random due to random seed used and passed into the Perlin noise function
2.2	Cellular Automata uses a randomly filled grid and iteratively collates points into tunnels
2.3	Trees are randomly generated on map generation
3.1	New games can be created on the server application
3.2	Players can connect through multiplayer
3.3	Title is displayed at the top, "Percy Dracott NEA"
3.4	Program can be closed through the windows "Min and Max" buttons
4.1	Up to 4 players can share a world at once
4.2	All updates to the world are synced across each client
4.3	Text chat allows players to interact and communicate within game
4.4	Players can connect through WAN using the servers private IP address, or through WAN by setting up the hosts router for Port Forwarding and using the public IP address of the server's computer
5.1	Worlds can be saved and loaded from the server application
5.2	Day and night are controlled by a light source which is moved up and down periodically
5.3	A zombie mob can attack the player, and is spawned randomly during the night
5.4	Both the Server and Client applications are Built for PC
5.5	Controls are optimised for mouse and keyboard, with it also being trackpad friendly with buttons.

Client Feedback

To assess whether my solution hits the brief given in the problem outline, I have asked my client to provide some feedback on the program, and whether it hits for potentially misses the brief.

Elliot: *"The game that you have made fits exactly the genre of creative sandbox, fitting right into the niche of complete sandbox I was requesting. As far as art style goes, the projects' graphics and design land on the exact style I was envisioning for the game."*

The map generation is very impressive, with an expansive cave system that is addictive to explore. Using the perlin noise for the landscape also gives the world's surface a really nice amount of variation with loads of open spaces to build on.

It is apparent that a great deal of attention was used to make the player interface of the game, including the player HUD. Of which I find is really well thought out for using with both a mouse or trackpad, as the large buttons lend themselves well to a trackpad.

As far as gameplay is concerned, the simple crafting and inventory system are easy to learn; in addition, the game is really enjoyable with friends.

Ideally, I think for more gameplay features, the game could do with a final goal, like a final boss or generated structure to be found. Perhaps within the caves as they are already designed for exploration."

Potential Improvements

Ideally, there are several parts of my program that I would like to change or implement, especially now after client feedback. Even though I did hit all my objectives.

A feature I would like to implement would be player inventory saving, a feature which I had planned out in Documented Design, perhaps using a relational database to store the players network ID as the primary key, username, and items.

As stated by Elliot a final event for the game would also be a feature to add, such as a castle or underground bunker that the player could fight through.

Source Tracker

- [1]. Riptide Networking - <https://riptide.tomweiland.net/manual/overview/about-riptide.html>
- [2]. Riptide Networking Basic Tutorial - <https://www.youtube.com/watch?v=6kWNZOFCQw>
- [3]. Cellular Automata Pseudocode - <https://www.youtube.com/watch?v=slTEz6555Ts&t>
- [4].