Bournemouth University

Faculty of Science and Technology

BSc (Hons) Games Software Engineering

May 2022

# *Utilizing Multiple Procedural Generation Techniques to Construct Organic 2-Dimensional Levels*

by

*Joseph Skinner*

Signed: JSKINNER

Name: Joseph Skinner

Date: 20 / 05 / 22

Programme: BSc GSE

## TABLE OF CONTENTS

## TABLE OF FIGURES

## ACKNOWLEDGEMENTS

Dedicated to my Gran Jen, who inspired me every day to be the best version of myself.

Thank you to my friends and family for supporting my research and participating in my qualitative research.

Primary Supervisor:

Simant Prakoonwit

BEng, MSc, PhD, DIC, MIEEE, MIET, FHEA

Associate Professor

Dept of Creative Technology, Faculty of Science and Technology

Secondary Supervisor:

Wen Tang

BSc, PhD, PGCE, MITE

Professor

Dept of Creative Technology, Faculty of Science and Technology

## ABSTRACT

“The pinnacle of game design craft is combining perfect mechanics and compelling fiction into one seamless system of meaning.”
(Sylvester, 2013)

Procedural Content Generation (herein referred to as PCG) has become a regularity in recent years. With such games as Minecraft and Terraria providing potentially endless replayability through the technique, it brings into question whether PCG will become the industry standard.

This portfolio project develops a survival game using Unity Engine that demonstrates the use of PCG to generate organic, top-down 2-Dimensional levels. Multiple techniques will be used in tandem to produce the game. The Overworld will be generated using Perlin Noise, and the caves will be formulated through several passes of Cellular Automata. These levels are populated with scenery and AI creatures for the player to interact with. The player can harvest resources, attack, and evade enemies, build structures, as well as craft useful items for their journey. All music has been composed specifically for the game by the writer.

Key milestones of the development process were evaluated, with self-reflection being used to refine the mechanics and gameplay. Later builds were released to participants to provide their own opinions and comments. The project’s end state is a completed game, the written dissertation that includes user feedback.

The writer has challenged themselves to completing this project as a way of exercising all their skills as a game developer as well as demonstrating the ability to deliver large pieces of work within a set time-frame.

## INTRODUCTION AND RATIONALE

In this portfolio project, the writer will develop a top down 2-dimensional survival game that utilises methods of PCG to generate the levels. They will examine the industry standard methods of PCG and review which techniques align best with the game's concept. Alongside this they will read into the generic conventions of the survival genre to implement the most pertinent features that encourages player immersion.

At the click of a button, games that utilise PCG features can generate an entire area of play space for the player to explore and interact with. If adapted and fine-tuned to suit the developer's desired aesthetic quality, this method can be a powerful tool that can save a lot of time and bring focus onto other segments of the game's creation. The aim of this project is to create a single-player survival game (provisionally titled, hosTILE) in which the player is thrown into a procedural world and forced to explore their surroundings, fend off against threats, and gather the resources necessary to escape.

As a Games Software Engineer, the writer's area of expertise is focused primarily on the programming and algorithmic side of game development. Being the lone developer on this project means that they cannot use time as an expendable resource to work on the level design and implementation of a large-scale survival level. Due to this, a PCG tool would be an excellent way to solve the looming problem others have in a similar situation, which is the creation of realistic, large-scale levels. Assigning the heavy lifting to the processor at runtime leaves me with time to refine and perfect the game mechanics which occur within these levels.

Within the games industry, there are many algorithms to produce procedural content, each of which having their own sets of advantages and disadvantages. For major companies or those working within strict timeframes, PCG can streamline the process of building levels, texturing models, or animations, and so reading into the subject will provide many advantages in the future of game development. The methods pursued in this paper are Perlin Noise for the overworld, and Cellular Automata for the caves. They both provide different visual outputs that are perfect for the scenarios in which they will be used. Perlin Noise creates expansive areas of terrain that can accommodate a multitude of biomes, whilst Cellular Automata creates snaking caverns that interconnect with one another. These two methods are not the only algorithms out there, however, and the review of literature will compare them against other worthy techniques like Worley Noise, Random Walk and Simplex Noise.

As stated previously, the aim is to create a survival game. During each significant step in the game's development the writer will release a build to a few select colleagues and friends to understand where the project's strengths and weaknesses lie. Testing is imperative in a game's development because positive audience reception and feedback provides data as to how to improve the product. Judging the success of a game alone is bad practise for future endeavours and will commonly leave the game feeling unpolished and, in some cases, not enjoyable.

## AIMS AND OBJECTIVES

The aim of this portfolio project is to **utilise PCG to create a survival game,** that uses the techniques of PCG so that a near infinite number of possible levels can be created. The game mechanics will allow the player to explore the level, harvest materials and resources, craft better equipment and enter conflicts with the NPC creatures that wander the play-space.
There are several objectives that are required to reach this goal, which upon compilation will result in a completed game. These are:

- **Quantitative research on PCG**
- **Research and implement a method to generate Perlin Noise and Cellular Automata**
- **Populate the world with resources**
- **Implement player mechanics**
- **Implement AI mechanics**
- **Implement items and functionality**
- **Beta Testing / Qualitative Research**

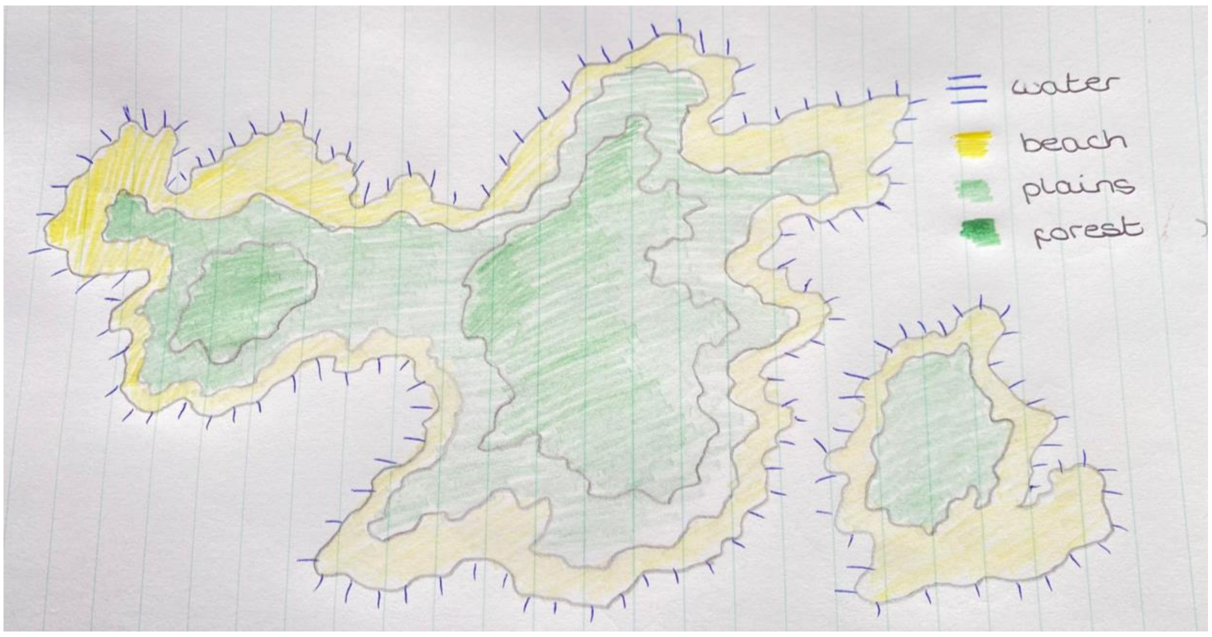


Figure 1 - Concept Art for Terrain Generation

## GENERIC CONVENTIONS OF THE SURVIVAL GENRE

**Figure 2 - Minecraft (Mojang, 2009)**

Survival games have been defined by Plarium.com (2019) as “*those in which you face off against a hostile environment in what is generally an open world, often starting with only meagre equipment, supplies and limited inventories*”. Linguazza.com (2014) builds upon this definition by adding, “Many survival games are based on randomly or procedurally generated persistent environments”. These procedurally generated worlds often mean that every playthrough of the game is drastically different. The player starts with nothing (or next to nothing) and must utilise the natural resources provided by the local habitat to obtain nourishment or better equipment.

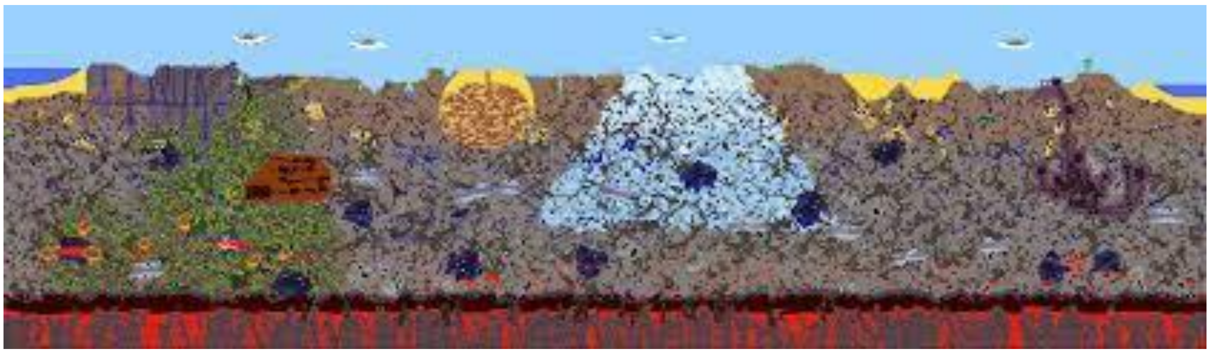


**Figure 3 - World Render (Terraria, 2011)**

For most titles within this genre, there is no defined end state. Instead, they take on a sandbox approach, in which the world is given to you, and is yours to manipulate and explore as you wish. The best-known example of this is Minecraft, where, although there are “quests” for the player to partake in, their completion does not decisively end the game. This idea will be taken into account and the gameplay will be based more on discovery and survival but will also have an end-game state where the player can build a boat to escape the island.

These conventions are a necessity for the project to be considered a survival game, so to begin the writer will research into the industry standard methods of PCG to create the world(s) of the game, before moving on to building the individual mechanics that will compile into the playable experience.

## REVIEW OF LITERATURE: PROCEDURAL GENERATION TECHNIQUES

### NOISE ALGORITHMS: PERLIN VS SIMPLEX VS WORLEY

In the context of a survival game, the terrain generation techniques are vital. Players will not feel as immersed in a world that is plain and / or repetitive, therefore this project will have a heavy emphasis on PCG methods to ensure that this does not happen. By relinquishing all control over the level for the system to handle both reduces production time and also provides a replayability factor that is not present with hand crafted, single levels (The Pros and Cons of Procedural Generation, 2022). Minecraft for example, uses a mixture of Perlin Noise and Perlin Worms when generating both the open spaces and the underground. This noise map is then given an additional feature to it that can better assign and spread the biomes, such as moisture or heat maps that reference real-world environmental models (Himite, 2021). For example, If the terrain height is low and the moisture level is minimal, the land will be generated as a desert.

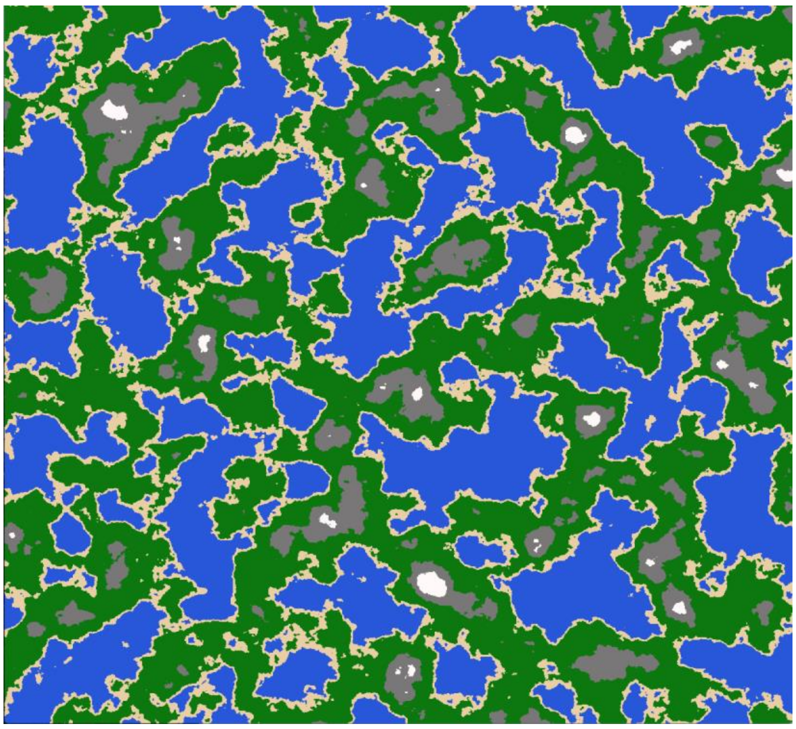


**Figure 4 - Example of a colour map generated by Perlin Noise (Scher, 2017)**

Perlin Noise is a very popular example of PCG. It is a procedural texture primitive that was developed by Ken Perlin as “A source of smooth, random noise” as he found generic random noise to be unfitting for some of its potential applications (Parberry, 2014). By defining a grid where every integer position in 3D space (Green, 2022) has a pseudo-random gradient vector, you can produce a render texture on which the shade of each pixel, going from 0 to 1, can represent the height map of a terrain topology. There does exist an alternative algorithm that is believed by many to be an improvement over Perlin Noise known as Simplex Noise. Again, developed by Ken Perlin in 2001, it addressed some of the issues that his original design had, such as its high computational complexity at the higher dimensions:

**O($n^2$)** in n dimensions instead of the **O($n^{2^n}$)** of Perlin Noise (Perlin vs. Simplex, 2022).

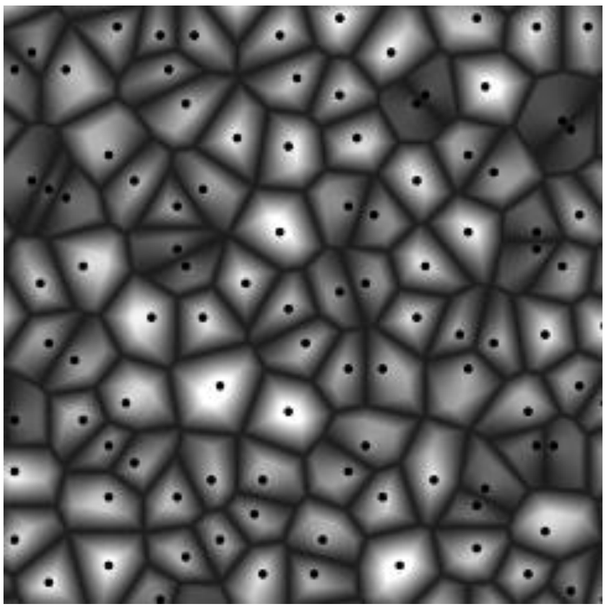


Figure 5 - Stefan Gustavson example of the application of Worley Noise)

Other methods of noise generation were considered as well. Worley noise is a primitive that produces good outputs, but said results are more fitting for biological modelling. Figure 5 shows an example of Worley Noise and its appearance is cell-like, which is not suited for the aim of this project. As mentioned by Gonzalez Vivo and Lowe (2022), Worley Noise has it's uses most prominently in the graphics and texturing community, and therefore would not produce results appropriate for that of the survival genre.
Perlin Noise has a lot of uses in games and other graphical areas but falls short when regarding cave generation. This is where cellular automata will come into play. "A cellular automaton consists of an nth dimensional grid with a number of cells in a finite state and a set of transition rules. Each cell of the grid can be one of several states; in the simplest case, cells can be on or off" (Pedersen, 2014). With this in mind, we have an opportunity to assign a set of predetermined rules to these cells to produce cave-like tunnels and open spaces.

## CAVE GENERATION: CELLULAR AUTOMATA VS RANDOM WALK

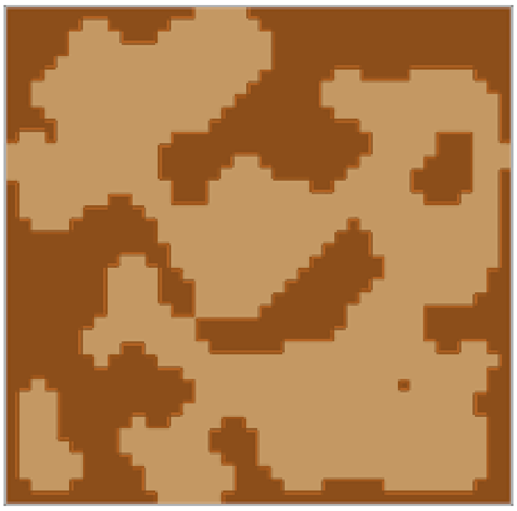


Figure 7 - A Cave Generated Using CA (Cook, Colton, Gow and Smith, 2019))

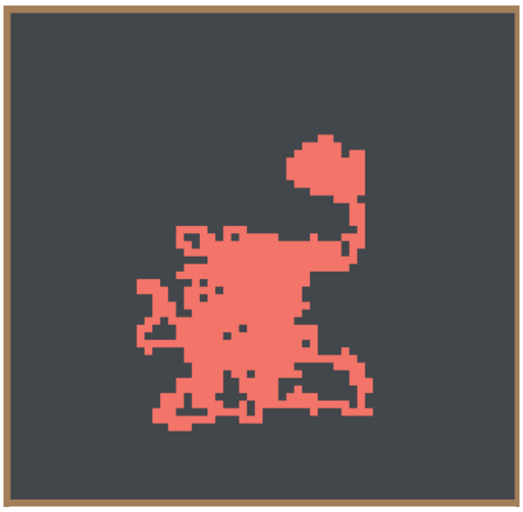


Figure 6 - Example of a cave generated with the Random Walk Algorithm (James, 2020)

Cellular Automata (herein referred to as CA) are abstract computational systems that use discrete steps to "evolve" (Cellular Automata (Stanford Encyclopaedia of Philosophy), 2017). The idea of CA was developed in the 1950s by multiple independent sources, but the most notable pioneer of the algorithm was John von Neumann, who endeavoured to create an abstract model of biological reproduction (Wolfram, 2002). The nature of CA means that they can be used to model complex and life-like outcomes, and within the context of game development, are perfect for generating scenes such as mazes and caves. The state of the cells can determine its role. If it is 1, then it is a wall, if it is 0, then it is open space, etc (see Figure 7). Random Walk algorithm is a procedure in which a defined number of randomly moving objects wander from their start position (Schmidt, 2022). The agents' movement is completely random, and by tracing their positions as time goes on the developer can produce a cave-like system. Unlike cellular automata, the use of RW guarantees that all pathways are accessible. Due to the nature of the algorithm, levels are generated using step limiters instead of a grid of defined size. This means that you cannot exactly control the size of the level produced as for the most part the agents will be moving back to positions, they have visited previously. This is likely to result in small levels that have little room to distribute natural resources. For a survival game where the theory of game balancing is vital, this is not a good approach.

## LITERATURE REVIEW CONCLUSION

Perlin Noise has ultimately been chosen over Simplex Noise because of three reasons. First, the world will be generated in its entirety at the start of runtime. This means that the algorithm halts before the player is introduced to the environment. The second reason is that the algorithm will only be generating in 2 dimensions, meaning that Perlin Noise's computational complexity will not have a noticeable effect on the gameplay. Simplex is much more computationally efficient as you increase the dimensions of noise. Since the game is 2 dimensional, these attributes do not need to be considered. The final reason is the writer's opinion. Perlin Noise's gradients are far more fitting for a natural environment, and so the implementation of such has been chosen. With the proposed level being a sparse and contained island, Perlin's biome generation method also organically separates the habitats to simulate their real world equivalents. It is unusual to find an island with a clear and obvious division between eco-systems, such as those produced with Worley Noise.

The project will use cellular automata in the generation of the caves ultimately due to its computational cost and the more interesting environments produced through the procedure. The use of CA also results in larger play-spaces, as it references and uses the defined dimensions of a grid compared to counting steps that are highly likely to go back on themselves multiple times. This can be seen in figure 6, where the cave-like structure generated is small in comparison to the surrounding area it could potentially fill. Also, these levels are being generated for use in the context of a survival game, therefore the narrow pathways and chaotic design of RW are not suited for player traversal and will confine all action through these avenues. This is poor game design and limits the choices a player has when it comes to enemy encounters, increasing the difficulty in these scenarios as well as removing aspects of meaningful play. Meaningful play is the relation between player action and the resulting outcome(s) due to them (Muñoz-Avila, n.d.). Restricting the play area translates to removing action possibilities from the player, taking away from the game experience

There is one drawback to CA against RW, and that is that some generated areas are inaccessible. This issue can be in a way corrected through optimising the birth and death rates of the cells, promoting growth over decline, however it is not guaranteed that a produced level will not carry any issues. Considering the aesthetic appearance of the entire system, however, makes the levels feel much more natural compared to RW.

## METHODOLOGY

### PLANNING

There are 3 stages to this project development:

1. Implement procedural generation algorithms
2. Create the game loop and mechanics
3. Test the game and use feedback to improve it

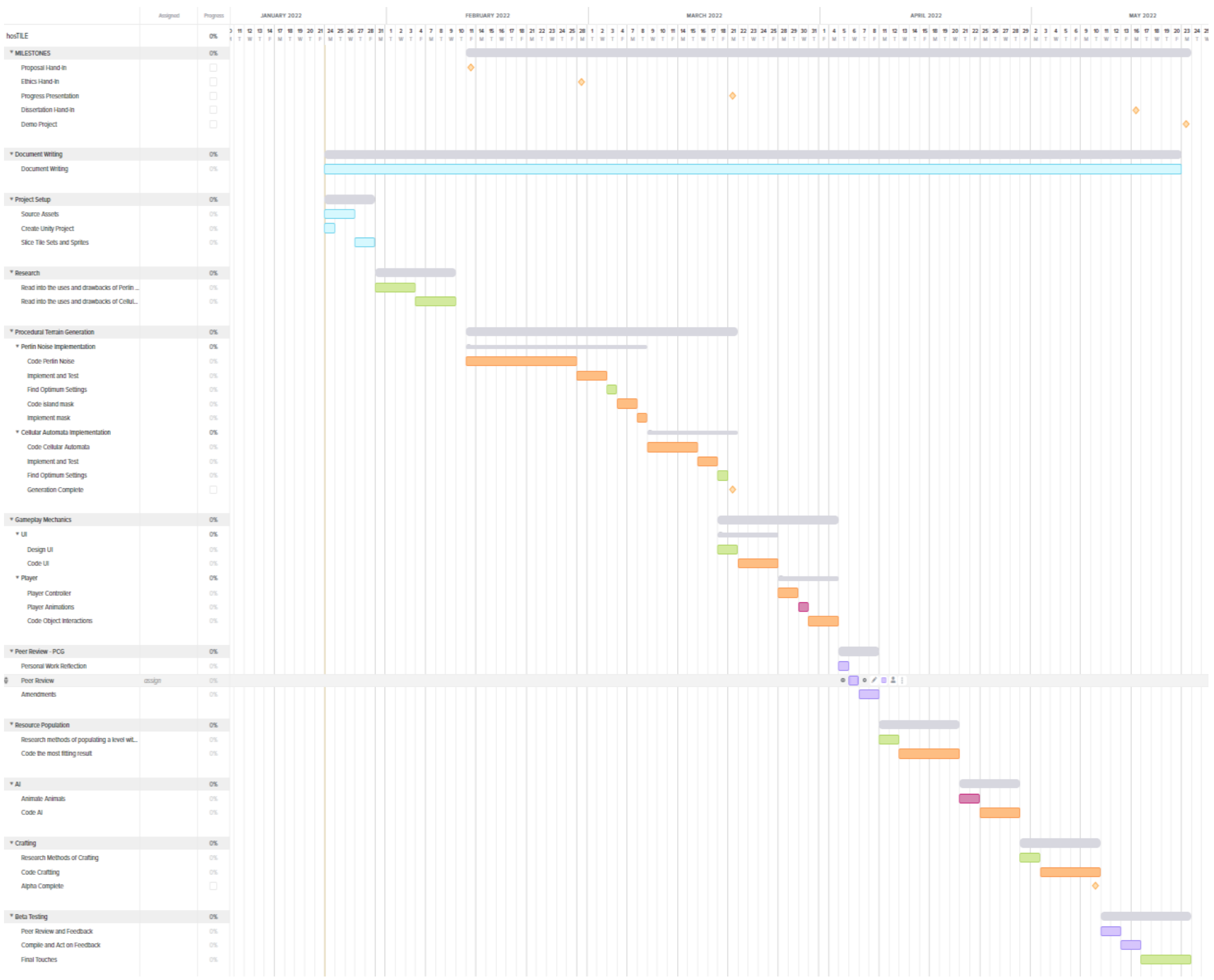


**Figure 8 - Project Plan Showing the Stages of Development to Create a Survival Game.**

The GANTT chart above was produced following the agile development methodology. This approach selects one objective to be focused in a "sprint", where the objective is completed within a set time frame. The terrain generation tools will be worked on first before the gameplay mechanics development begins. This has advantages as this area of the project is estimated to be the costliest in terms of time and research which means that completing this sooner will reduce the weight of any crunch developing nearing the project's completion. Secondly, having steady and reliable generation framework eases the advancement of the mechanics because there is a solid level in which to test and evaluate each step. Note that the writing of this document is worked on concurrently to the project.

## SOFTWARE

The game will be developed using Unity 2019.4.14f1 due to the writer's experience with the IDE and their proficiency using c# programming language. Unity offers many tools and utilities that would take a significant amount of time to develop from scratch if one was to create an entirely self-made engine. This allows a developer to focus on the design and implementation of the game over the construction of the engine, saving valuable time and expendable resources as well as focusing efforts on the project at hand.

## GRAPHICAL AESTHETIC AND USE OF THIRD-PARTY ASSETS

The game's graphics will consist of 16x16 sprites and tile sheets. These graphics are taken from the same creator to keep aesthetic congruence [see Third Party Assets Section in Appendix]. The tile sheet includes different types of water, beach and sand tiles, grassland variations, and mountains. Pixel sprite art was chosen due to its simple and friendly appearance. The objects are clearly visually defined, and the landscape can transition from one habitat to the next (see Figure 9).

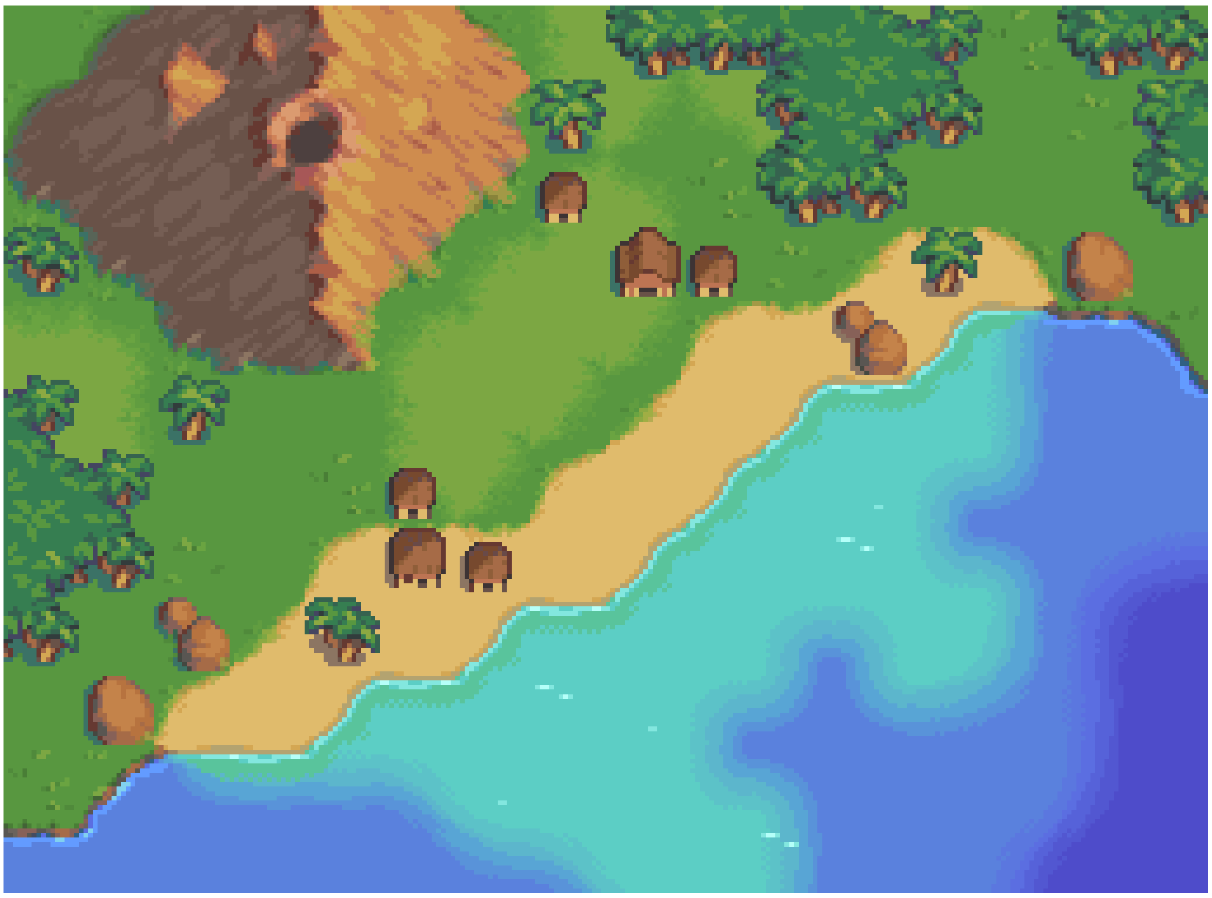


**Figure 9 - Example Image from Asset Pack (Pita, 2018)**

## GAME FLOW AND DESIGN

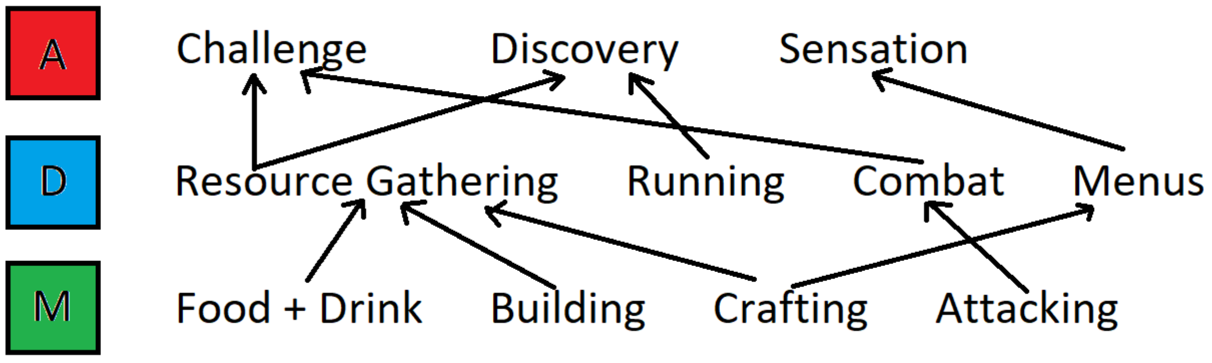


Figure 10 - MDA Diagram of hosTILE

The idea of this project is for it to be a short, simple survival game, that can be played through to completion within 30 minutes. The overworld will be large and diverse enough that the player is encouraged to explore. The caves are single-room levels embedded within the overworld and will be full of winding paths and more hostile NPC creatures. To complete the game (if they wish), the player must search every part of the island to gather the necessary resources to build a raft and escape.

## MENU FLOW

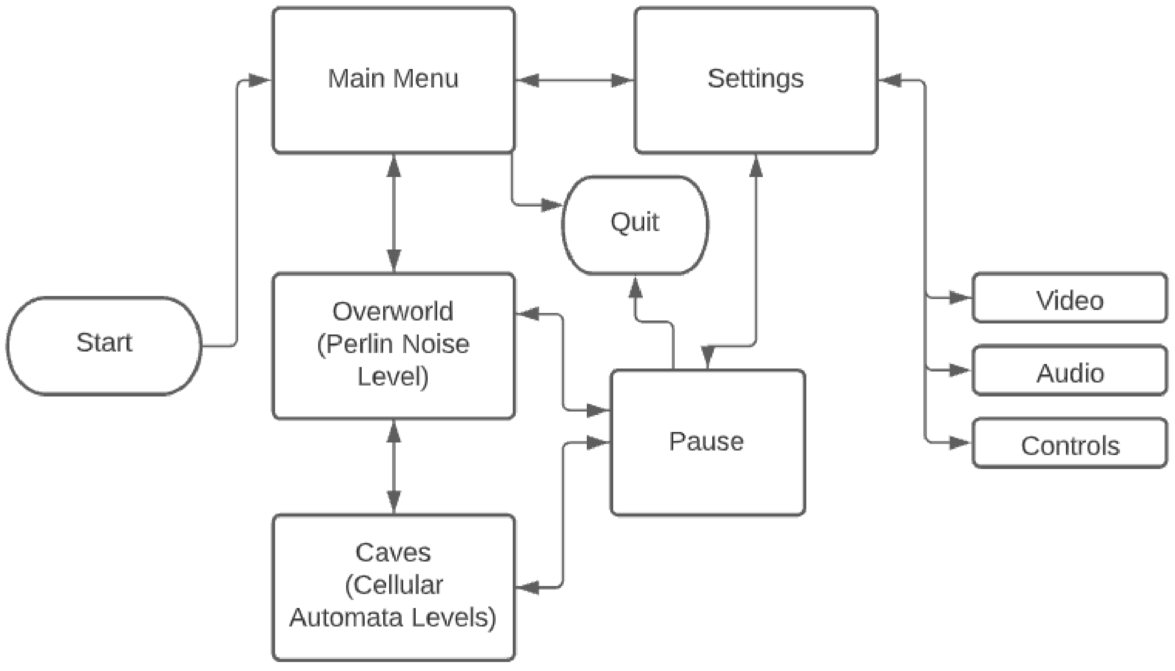

Main Menu
Settings
Quit
Start
Overworld (Perlin Noise Level)
Pause
Video
Audio
Controls
Caves (Cellular Automata Levels)

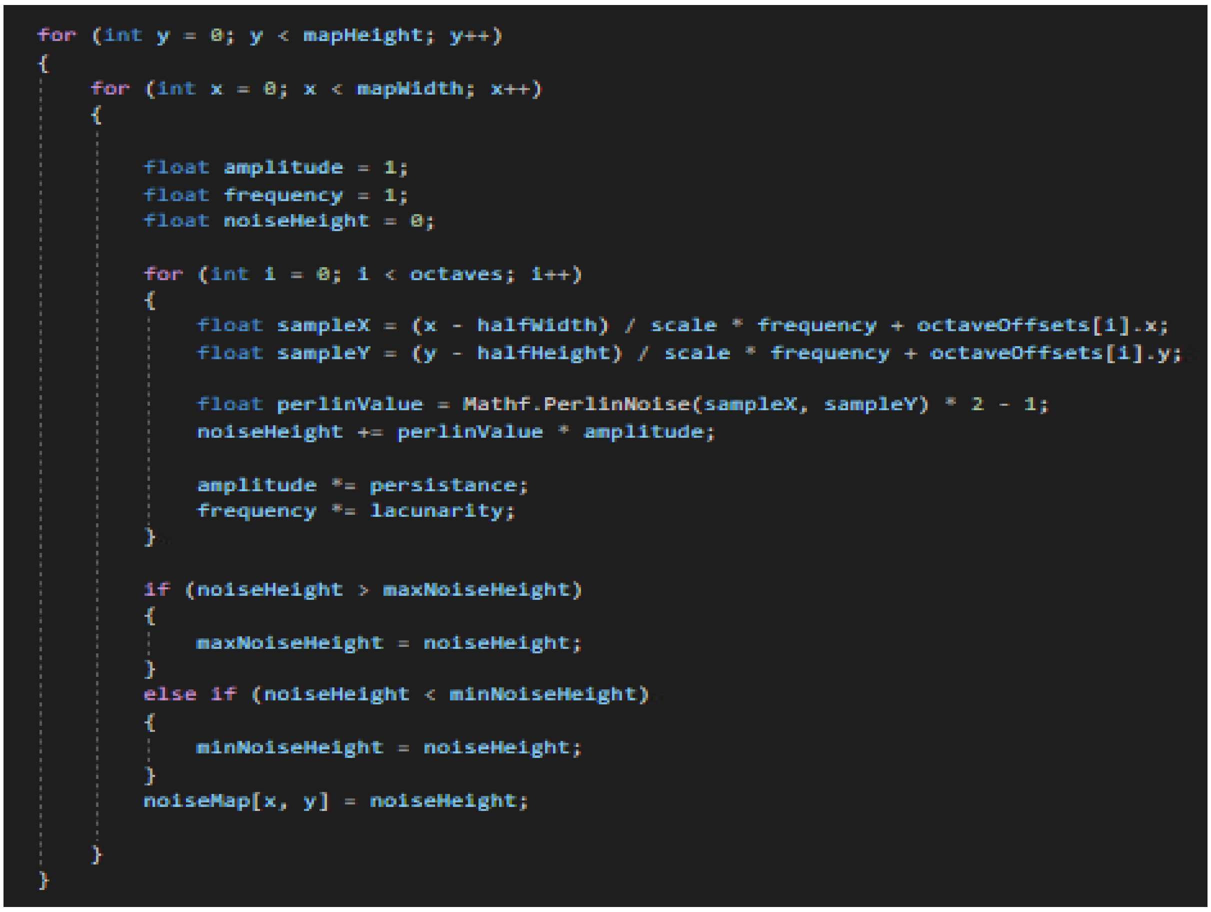
Figure 11 - hosTILE Menu Flow Diagram (Lucidchart)

## IMPLEMENTING PERLIN NOISE

### GENERATING NOISE

Unity Engine has a built-in Perlin Noise Function that is utilised within this project. It is used to streamline the development process of the game.

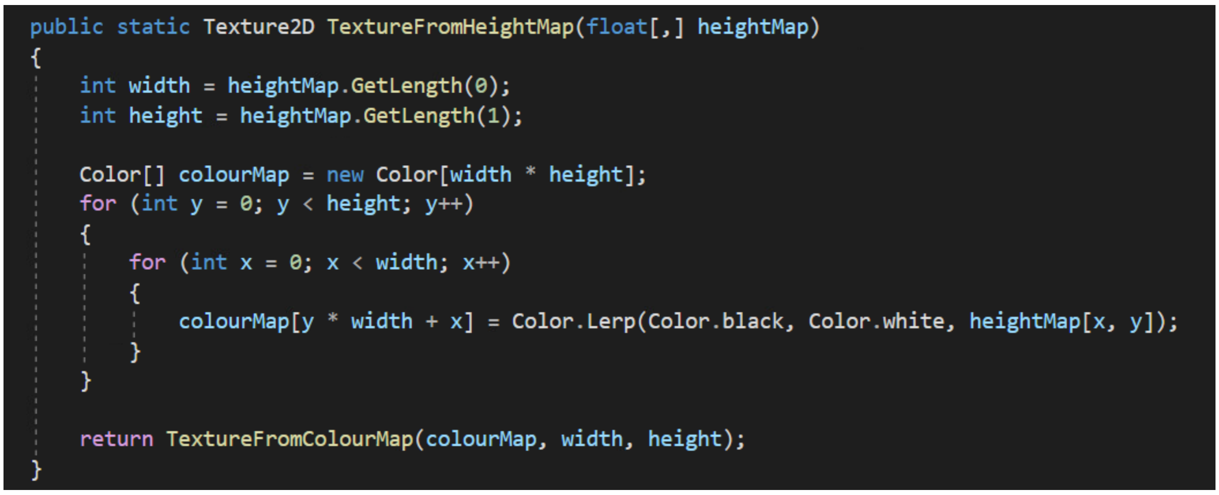
Every cell in the grid is assigned a pseudo-random number, meaning that the value 0-1 is random yet its magnitude remaining relatively close to that of the values around it. Without the use of pseudo-random numbers, the resulting image would be a mess of random coloured pixels, and the resulting terrain would have no natural flow or congruency. These numbers are then interpolated using rules that are set within the script.



```
for (int y = 0; y < mapHeight; y++)
{
    for (int x = 0; x < mapWidth; x++)
    {

        float amplitude = 1;
        float frequency = 1;
        float noiseHeight = 0;

        for (int i = 0; i < octaves; i++)
        {
            float sampleX = (x - halfWidth) / scale * frequency + octaveOffsets[i].x;
            float sampleY = (y - halfHeight) / scale * frequency + octaveOffsets[i].y;

            float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;
            noiseHeight += perlinValue * amplitude;

            amplitude *= persistance;
            frequency *= lacunarity;
        }

        if (noiseHeight > maxNoiseHeight)
        {
            maxNoiseHeight = noiseHeight;
        }
        else if (noiseHeight < minNoiseHeight)
        {
            minNoiseHeight = noiseHeight;
        }
        noiseMap[x, y] = noiseHeight;

    }
}
```

**Figure 12 - Code Snippet producing Perlin Noise**

Each cell's gradient is then used to shade the corresponding pixel on the render texture. The outputted image shows smooth transitions from white to black. Although it is greyscale, the result is open for further manipulation. Perlin Noise uses "octaves", which are functions that determines the level of detail produced in the noise map by layering multiple fractals (World-Machine.com, 2022). In this project, the optimum number of octaves is 6. Now that the noise map has been produced the next step is to introduce boundaries that separate the heights into biomes. Using the Unity Editor functionality, the Perlin Noise script can be changed into an interactive tool that can organise the height boundaries. This example has the lowest bounds coloured blue to represent deep water, with lighter blue reflecting shallow waters. Beaches are followed (yellow) and then two grassland biomes (light and dark green). Now, what used to be random noise is beginning to look more like terrain from a bird's eye perspective. This is still just an image however, so the theory behind this is taken to a tile map. A tile map is a tool within Unity that uses cell grids to store sprite images. These images are generally level tiles that build the aesthetic of a level within 2-dimensional games, and can be animated. The Perlin Noise map produced can be manipulated and amended changing the *Lacunarity* and *Persistence* values. Lacunarity refers to the increase in frequency which in turn increases the detail. Lacunarity over the value of 1 will increase the noise quality through every layer. Persistence refers to the strength of which the layers of noise are reduced through every successive octave.



```
public static Texture2D TextureFromHeightMap(float[,] heightMap)
{
    int width = heightMap.GetLength(0);
    int height = heightMap.GetLength(1);

    Color[] colourMap = new Color[width * height];
    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            colourMap[y * width + x] = Color.Lerp(Color.black, Color.white, heightMap[x, y]);
        }
    }

    return TextureFromColourMap(colourMap, width, height);
}
```

Figure 13 - Code Snippet Translating Noise Map to Colour Map

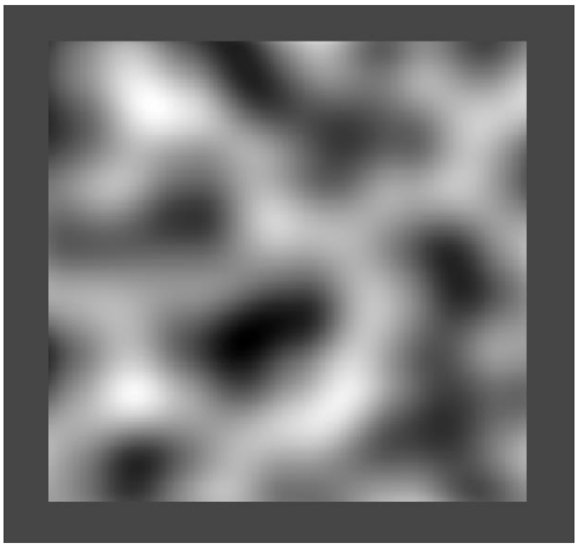


Figure 15 - Perlin Noise Generation Stage 1

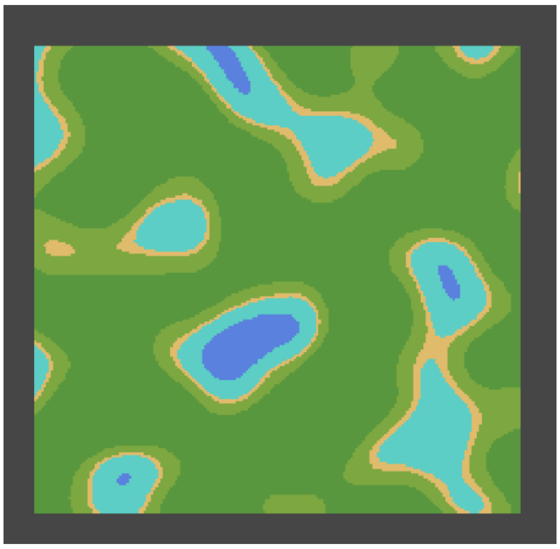
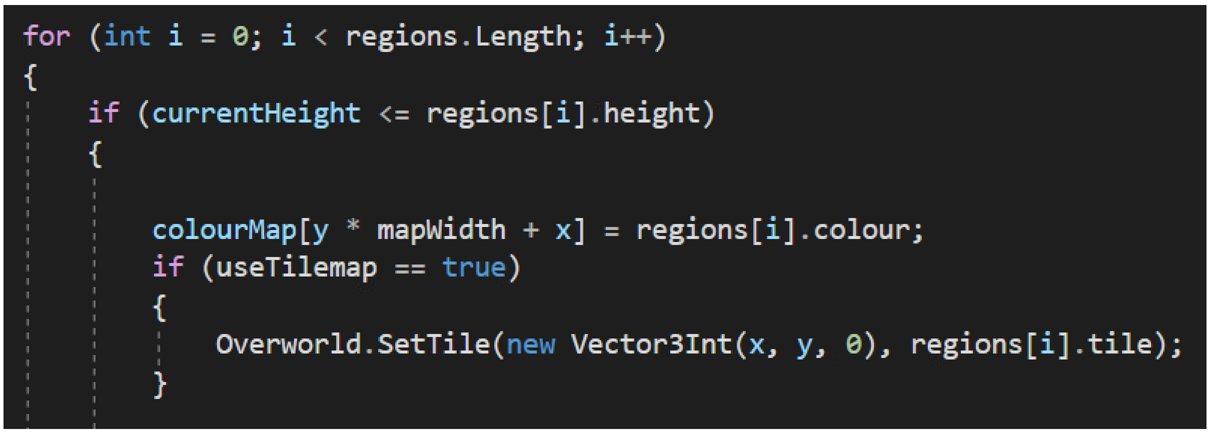


Figure 14 - Perlin Noise Generation Stage 2

## TRANSLATING NOISE TO TILE MAP

The framework for the level generation is now in place, so the next step is to we introduce a tile map. Now instead of changing the colour of a pixel, we paint the corresponding biome tile into the tile map grid.

To ensure that the tiles fit with their neighbours, we use Unity's built-in "Rule-Tile" tool. These are scriptable tiles that can change their sprite image according to which tiles are adjacent to them.



```
for (int i = 0; i < regions.Length; i++)
{
    if (currentHeight <= regions[i].height)
    {

        colourMap[y * mapWidth + x] = regions[i].colour;
        if (useTilemap == true)
        {
            Overworld.SetTile(new Vector3Int(x, y, 0), regions[i].tile);
        }
```

Figure 16 - Code Snippet Translating Colour Map to Tile Map

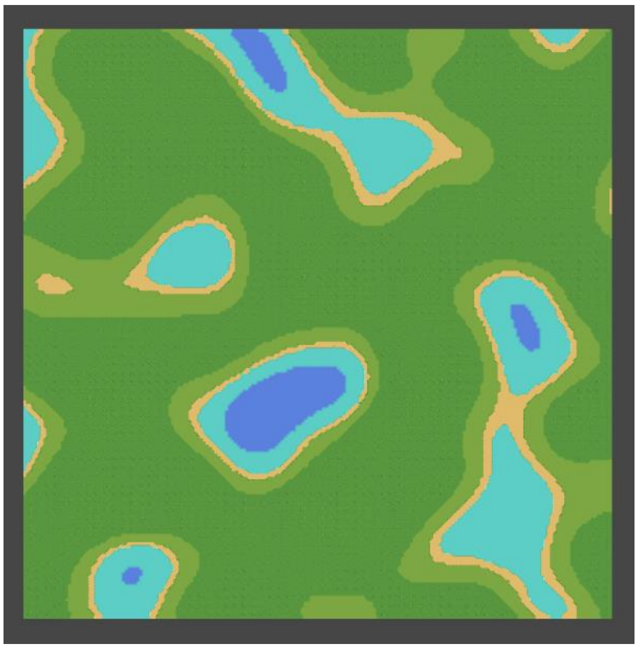


Figure 17 - Perlin Noise Generation Stage 3

## OPTIMISING BIOME SETTINGS

The sizes and shapes of each biome are a large part of world building. It is important that each habitat in the game reflects their real-world counterpart. For example, beaches and coasts typically line the meeting-point between the sea and the land as a thin strip. We need to amend the settings of the Perlin Noise editor window to ensure the output is generated as such. The vision of the grasslands and forests is for the lighter pastures to be expansive, and host to a lot of bushes and shrubs, and populated with few trees. The darker forest spots are the inverse, being smaller and containing many trees, and little in terms of small plant life.

| **Biome** | **Objects Found** | **NPCs Found** | **Height Ranges** |
|---|---|---|---|
| Caves | Rocks (Abundant)<br>Iron Ore (Few) | Slimes | N/A |
| Deep Water | N/A | N/A | 0 |
| Shallow Water | Clams (Rare) | Crabs<br>Slimes | 0 - 28 |
| Beach | Rocks (Few) | Crabs | 29 - 42 |
| Grassland | Trees (Few)<br>Bushes (Abundant) | Rabbits | 43 - 54 |
| Forest | Trees (Abundant)<br>Bushes (Abundant) | Rabbits<br>Birds | 56 - 100 |

Figure 18 - Table to Show Biome Data

## SEEDS

A seed is a numerical value that acts as a unique ID for a level. This number can be input into the Perlin Noise Generator to reliably reproduce the terrain if the player wishes to replay a level (Preston, 2018). The code for the generator includes the option to randomise the seed as well as the offsets of the noise map, which results in a completely new level upon generation. The upper limit of seeds that can be produced is 9.9 * 10^5, and the offsets for both x and y is 500. This means that the total possible levels that can be created is:

(9.9 * 10^5) * (500^2) = **2.49 * 10^10 potential levels**

## ISLAND MASK

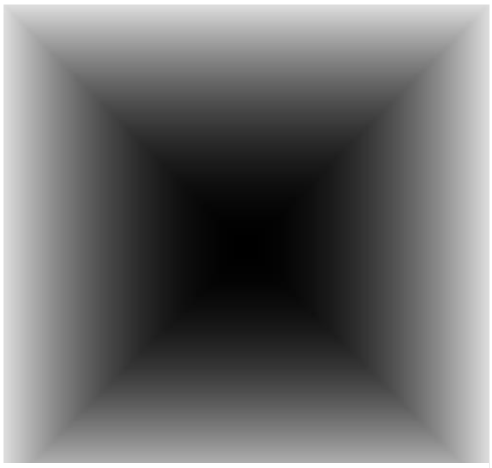
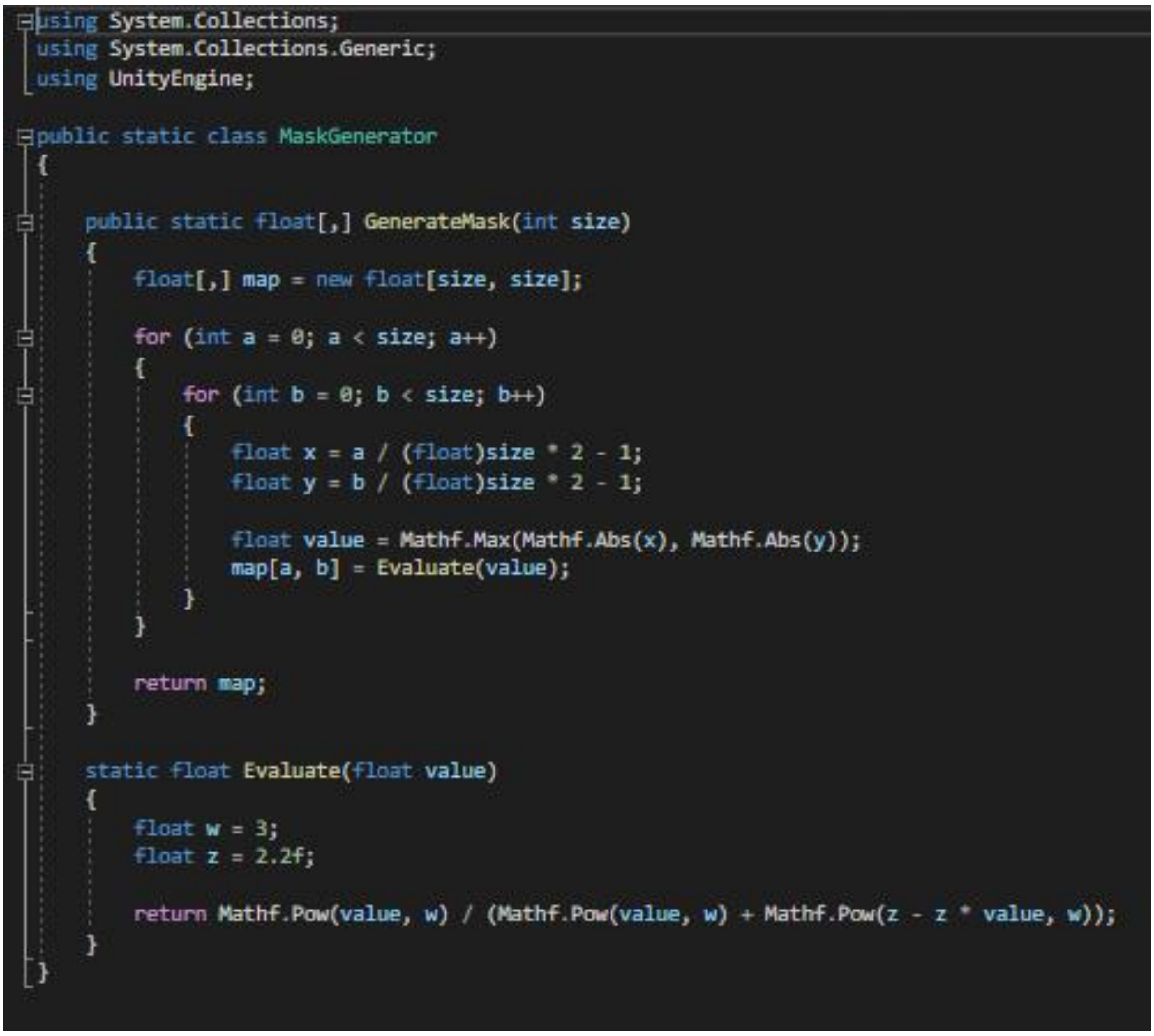


Figure 19 - Square Gradient (Travall, 2018)

Now the generation algorithm successfully produces a landscape with biomes, however the grid size means that at the borders, the landscape is abruptly cut off. We need to confine the play-space onto an island to stop said landscape spilling out of the grid's view. This requires the use of a square gradient mask.

Figure 19 shows a square gradient. Its values are subtracted from those of the noise map produced earlier to reduce the outer edges to zero. In the game, the value of 0 is translated to deep water, creating an island in the centre. This is a necessity if we want our game to have a clear and defined shape and bounds.



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

public static class MaskGenerator
{

    public static float[,] GenerateMask(int size)
    {
        float[,] map = new float[size, size];

        for (int a = 0; a < size; a++)
        {
            for (int b = 0; b < size; b++)
            {
                float x = a / (float)size * 2 - 1;
                float y = b / (float)size * 2 - 1;

                float value = Mathf.Max(Mathf.Abs(x), Mathf.Abs(y));
                map[a, b] = Evaluate(value);
            }
        }

        return map;
    }

    static float Evaluate(float value)
    {
        float w = 3;
        float z = 2.2f;

        return Mathf.Pow(value, w) / (Mathf.Pow(value, w) + Mathf.Pow(z - z * value, w));
    }
}
```

Figure 20 - Code Snippet Showing the Square Gradient Algorithm

Subtracting the gradient from the generated Perlin Noise creates a squared fall-off around the perimeter of the noise map which “pushes” the values of each pixel to zero. When it comes to colour map, these reduced values mean that the terrain is surrounded by water, creating a confined play-space for the game-loop to take place.

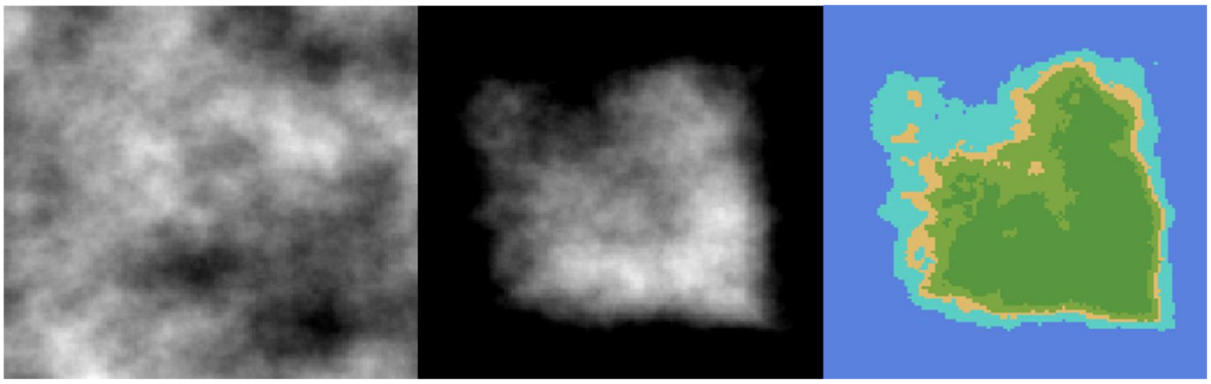


Figure 21 - Step-By-Step Process to Generate a Perlin Island

## FINAL RESULTS

After producing the Perlin Noise map, subtracting the square gradient, and translating it to a tile map we are left with a level produced solely through scripting. The shape and biome distribution looks natural for a desert island of that size, and these organic looking results can be reliably reproduced with the use of different seeds. Figure 22 shows how the game level has similarities to real world islands, especially with regards to the surrounding water and greenery.

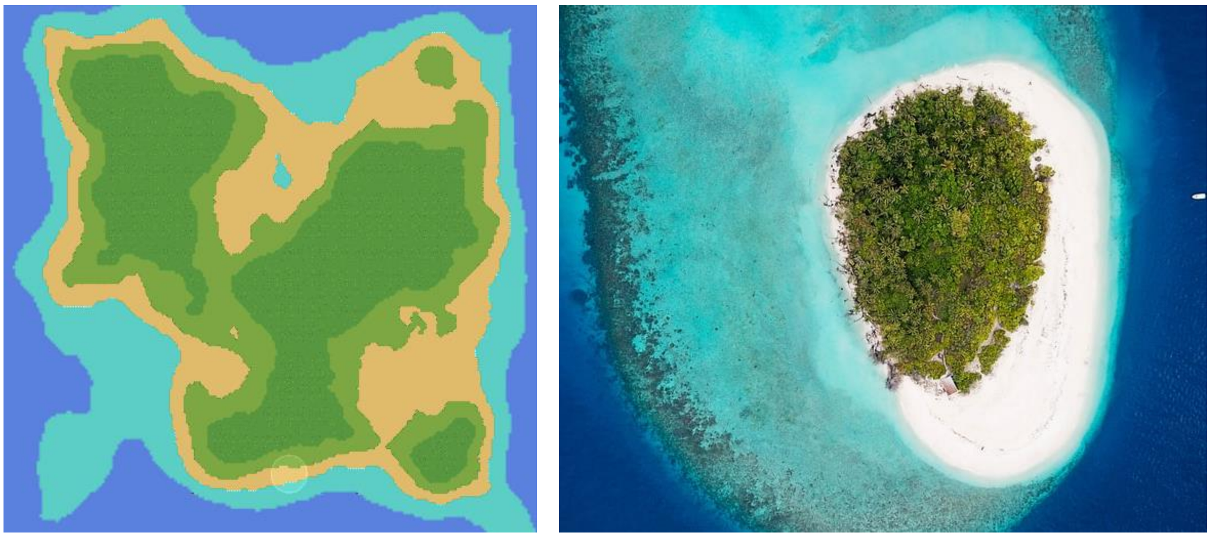
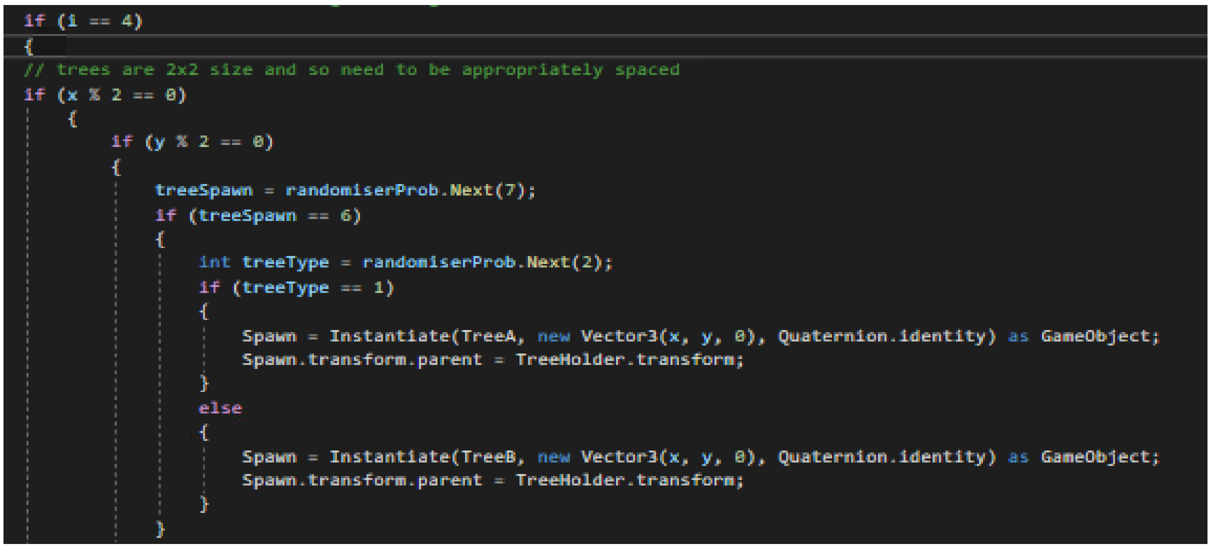


Figure 22 - Perlin Noise Island Generation and a Real-World Desert Island (Ahmed, 2018)

## RESOURCE DISTRIBUTION

At this moment the landscape is being produced however there are no resources within. Using the height maps we can instantiate the correct objects for the environment to fill this empty space.  For example, if the program is placing forest tiles, there is a chance that a tree will be placed onto said tile. Each biome will have different natural resources that can spawn.



```
if (i == 4)
{
// trees are 2x2 size and so need to be appropriately spaced
if (x % 2 == 0)
    {
        if (y % 2 == 0)
        {
            treeSpawn = randomiserProb.Next(7);
            if (treeSpawn == 6)
            {
                int treeType = randomiserProb.Next(2);
                if (treeType == 1)
                {
                    Spawn = Instantiate(TreeA, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                    Spawn.transform.parent = TreeHolder.transform;
                }
                else
                {
                    Spawn = Instantiate(TreeB, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                    Spawn.transform.parent = TreeHolder.transform;
                }
            }
```

Figure 23 - Snippet from Resource Distribution Program (Trees)

The world generator is producing good levels each time, and the resources are issued onto the terrain dependant on the height level. Through every height check a random number is produced and compared to a hard coded integer. If these two variables are equal, then the natural resource for that height level is instantiated onto the level on the same coordinate as the tile placed before it. This will result in a level with resources completely randomly assigned across the landscape.

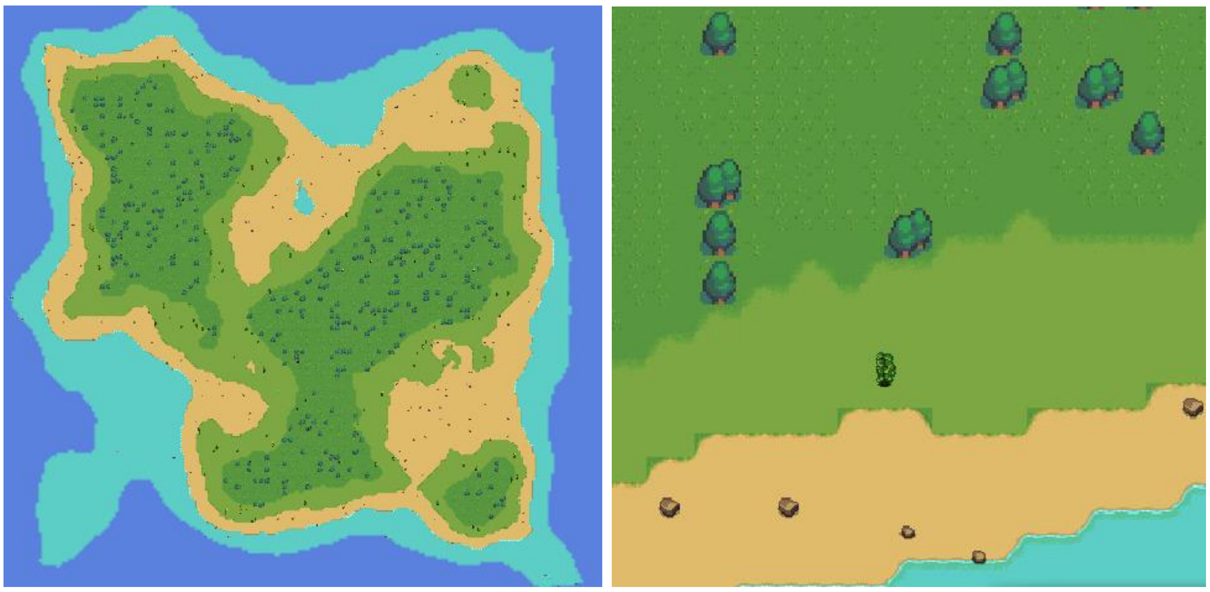


Figure 24 - Far and Near Shots After Resource Distribution

## IMPLEMENTING CELLULAR AUTOMATA

As mentioned in the review of literature, cellular automata are discrete systems that populate a grid depending on a set of rules declared by the developer. The rules that will be set will follow the theory of "Conway's Game of Life", a demonstration of the growth and decline of organic-like structures held within the grid of cells (Caballero, Hodge, and Hernandez, 2016).

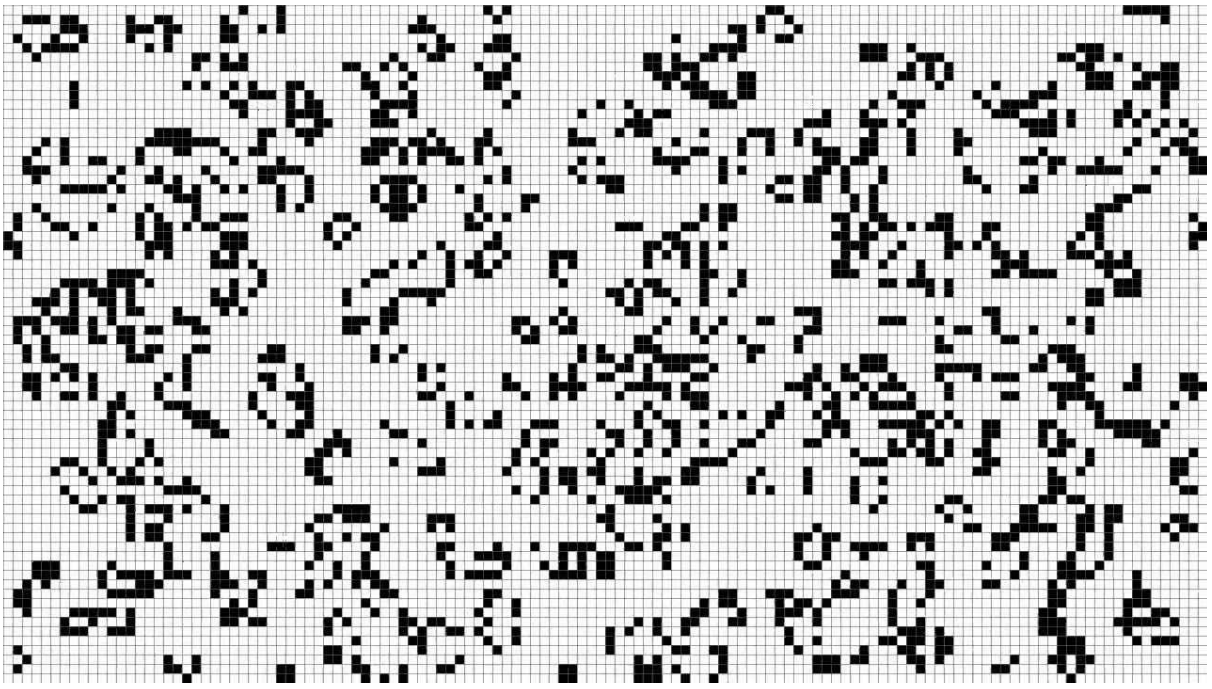


Figure 25 - Example of Conway's Game of Life (Bettilyon, 2018)

## DEVELOPING CELLULAR AUTOMATA

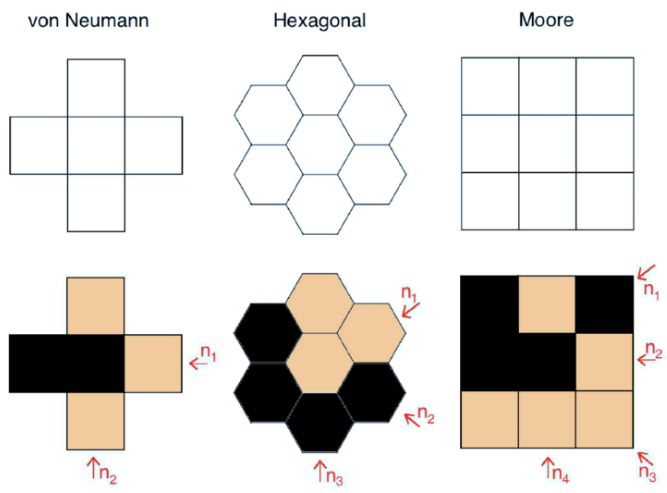
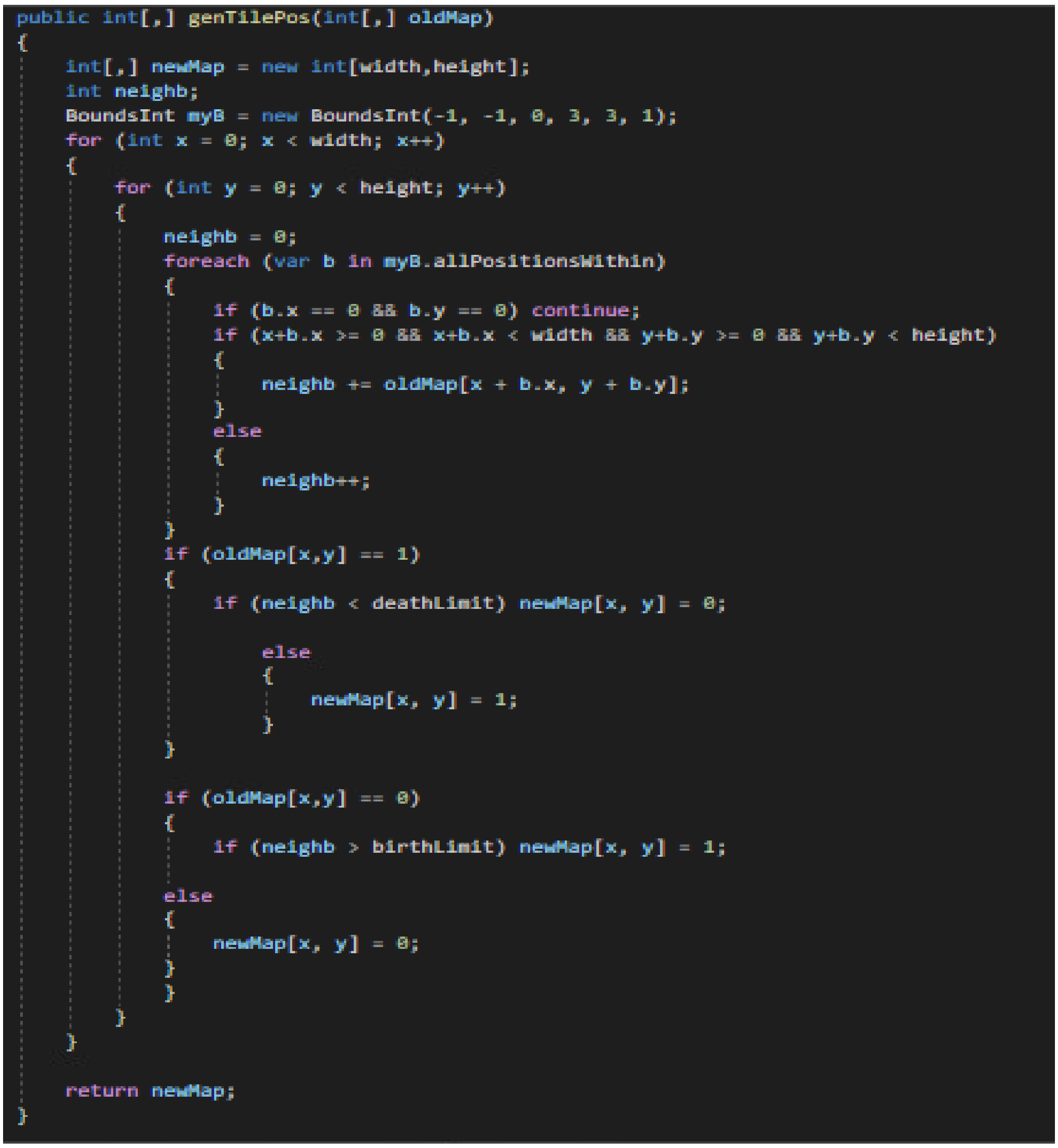


Figure 26 - Symmetrical Von Neumann, hexagonal and Moore neighbourhoods in two dimensions (García-Morales, 2012)

Conway's Game of Life first sequentially sets each cell in the grid as either 1 or 0, which represents its contents being alive or dead. In the code, this probability variable is set to 50%. With this in place, the grid should be half populated with alive cells, and now the generation procedure can begin.

Two more parameters are declared that are known as the **birth limit** and the **death limit**.

The user developing the CA can manipulate the values of these from 1-8 to produce different results on the grid. There are two significant “neighbourhoods” that can be used within this section of the algorithm, which check different cells in proximity. The first is the “Moore Neighbourhood”, where the adjoining cells are the only deciding factors The second is the “Von Neumann Neighbourhood”, which contrasts against neighbours in contact with the vertices. This project will use Von Neumann’s neighbourhood. Each active (alive) cell will have its dead neighbours checked and contrasted against the death limit. If the number is lower than the parameter set, the cell will die. All inactive (dead) cells are contrasted against the birth limit, and if there are more alive neighbours than there dead, the cell will become active (Kowalski, 2020). This process is iterated upon multiple times and with each loop, the cave system will begin to take shape. In this simulation, the perfect number of loops was discovered to be **4**.



```
public int[,] genTilePos(int[,] oldMap)
{
    int[,] newMap = new int[width,height];
    int neighb;
    BoundsInt myB = new BoundsInt(-1, -1, 0, 3, 3, 1);
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            neighb = 0;
            foreach (var b in myB.allPositionsWithin)
            {
                if (b.x == 0 && b.y == 0) continue;
                if (x+b.x >= 0 && x+b.x < width && y+b.y >= 0 && y+b.y < height)
                {
                    neighb += oldMap[x + b.x, y + b.y];
                }
                else
                {
                    neighb++;
                }
            }
            if (oldMap[x,y] == 1)
            {
                if (neighb < deathLimit) newMap[x, y] = 0;

                    else
                    {
                        newMap[x, y] = 1;
                    }
            }

            if (oldMap[x,y] == 0)
            {
                if (neighb > birthLimit) newMap[x, y] = 1;

            else
            {
                newMap[x, y] = 0;
            }
            }
        }
    }

    return newMap;
}
```

Figure 27 - Code Snippet for Birth and Death Rates

Just like the implementation of Perlin Noise, we are substituting this theoretical grid for a tile map. The state of the cell decides whether it will be a wall tile or a floor tile. Wall tiles are given a box collider, so the player is limited to walk through the open areas only.

The examples below show the functionality of the birth and death rates. Through each iteration, every cell is re-evaluated following the declared rules and over time, forms the level on which the player will have their experience. Just like the Overworld, this scene will be populated with resources for the player to harvest.

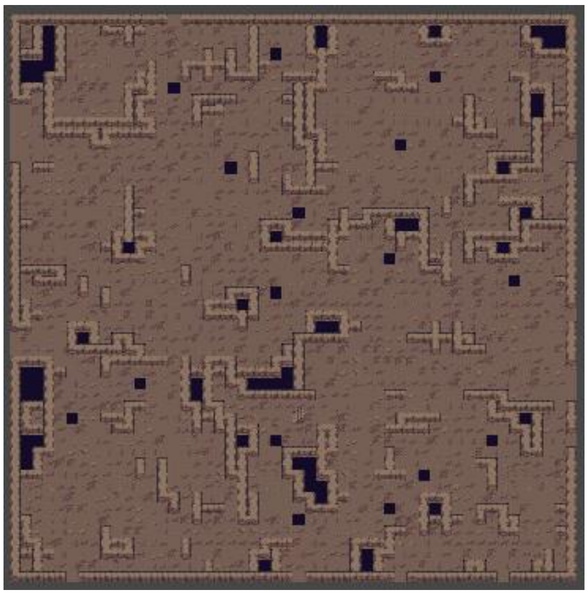


Figure 29 - CA Iteration 1

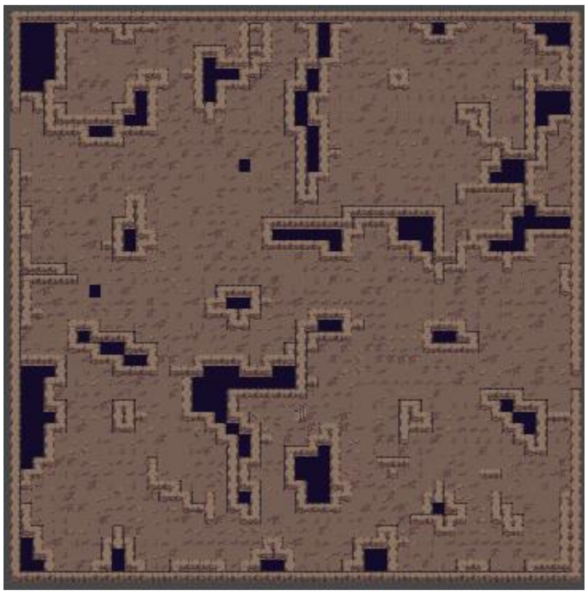


Figure 30 - CA Iteration 2

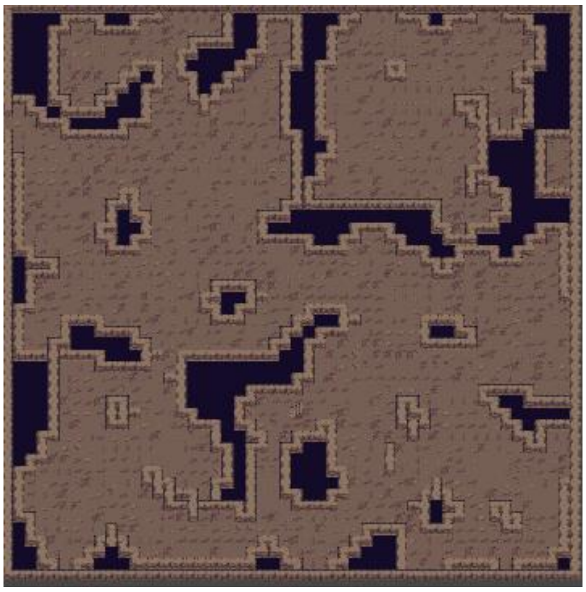


Figure 28 - CA Iteration 3

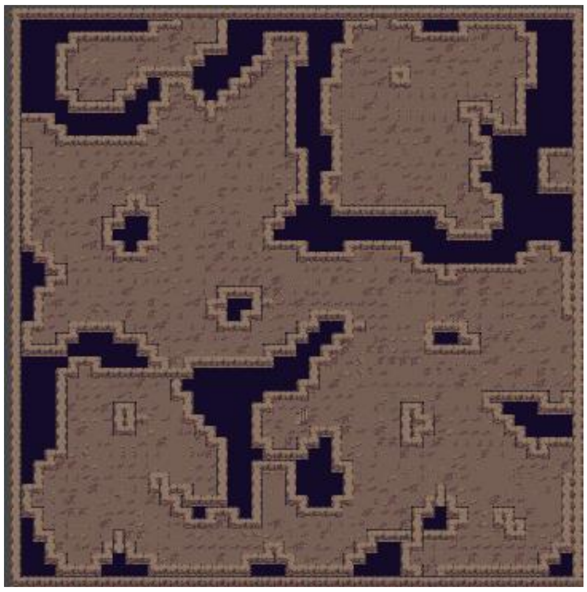


Figure 31 - CA Iteration 4

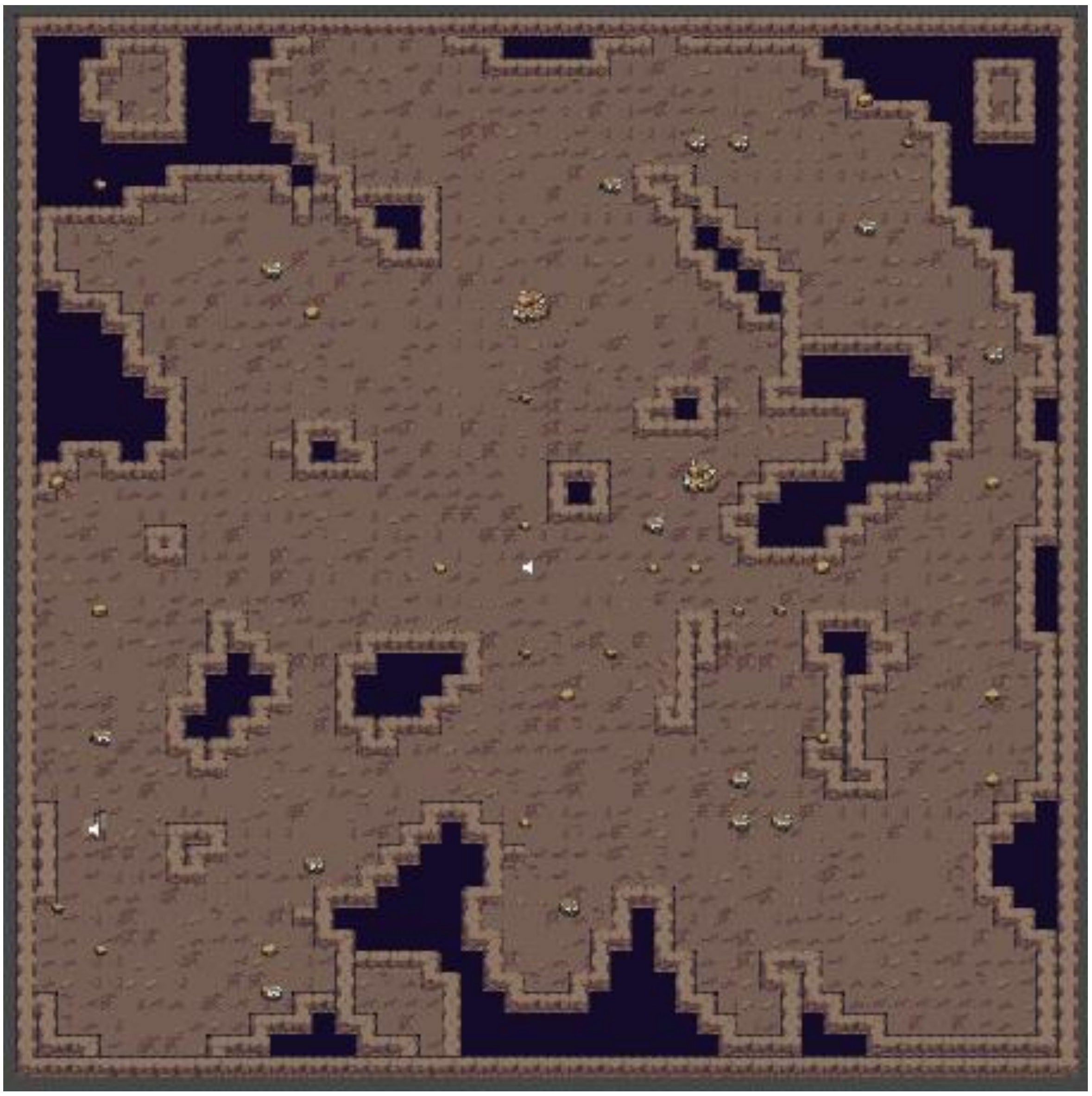


Figure 32 - Example of CA and Resource Population

With the algorithm working up to standard, all that remains is to create an entry point between the overworld and the caves. In "hosTILE", these take the form of a small, raised bit of land with an entrance, and a stone staircase. Being within range of these objects will prompt the player to interact with them, which then sets their new position to somewhere else depending on whether they are entering or leaving the cave. These objects are placed into the resource allocation section of their respective algorithms with instantiation rules and are therefore randomly placed into their environment.

## IMPLEMENTING THE PLAYER

The player avatar is the vessel through which the user enters the magic circle and interacts within the game's mechanics. The player has some strict requirements that need to be met for them to survive in the game:

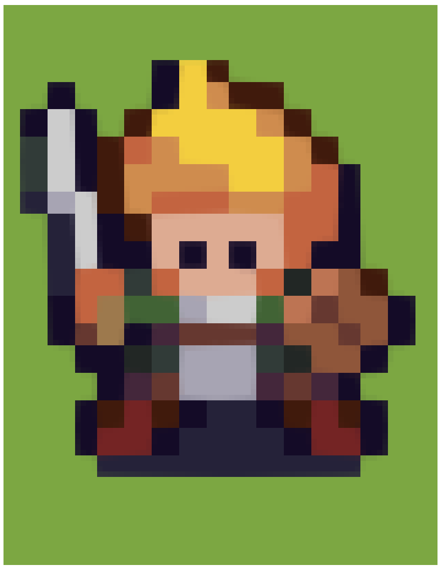


Figure 33 - hosTILE Player Character

- The player must drink enough water. The water bar cannot fall below zero or the health bar will reduce drastically and constantly. To stay hydrated, the player must visit the shallow water to drink, or kill slimes that drop water bottles.

- The player must stay healthy. Taking damage from enemies will whittle down the player's health until it reaches zero, where the game will end. To keep the health up, the player is required to eat food.

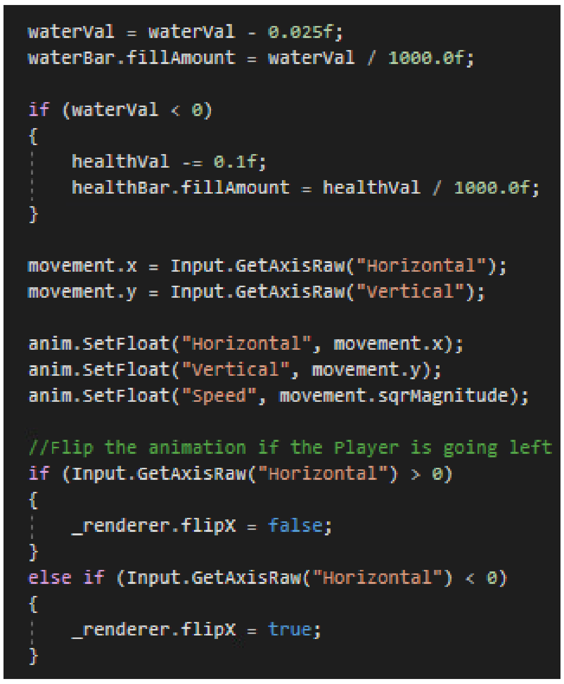
- The player should harvest the resources and craft better equipment that will ease the process of gaining more resources / escaping. Wood, rope, iron, and pearls are the four constituents of the crafting system and combining them in different ways will produce different results.

### PLAYER CONTROLLER

The player controller is a series of scripts that governs how the player interacts with the world. This ranges from the movement of the avatar, the handling of animations and game mechanics (i.e., attacking, picking up items, etc). and transporting the player from the overworld to the caves and back. The player controller also keeps a track of the state of the player, like their water and health values.

### MOVEMENT

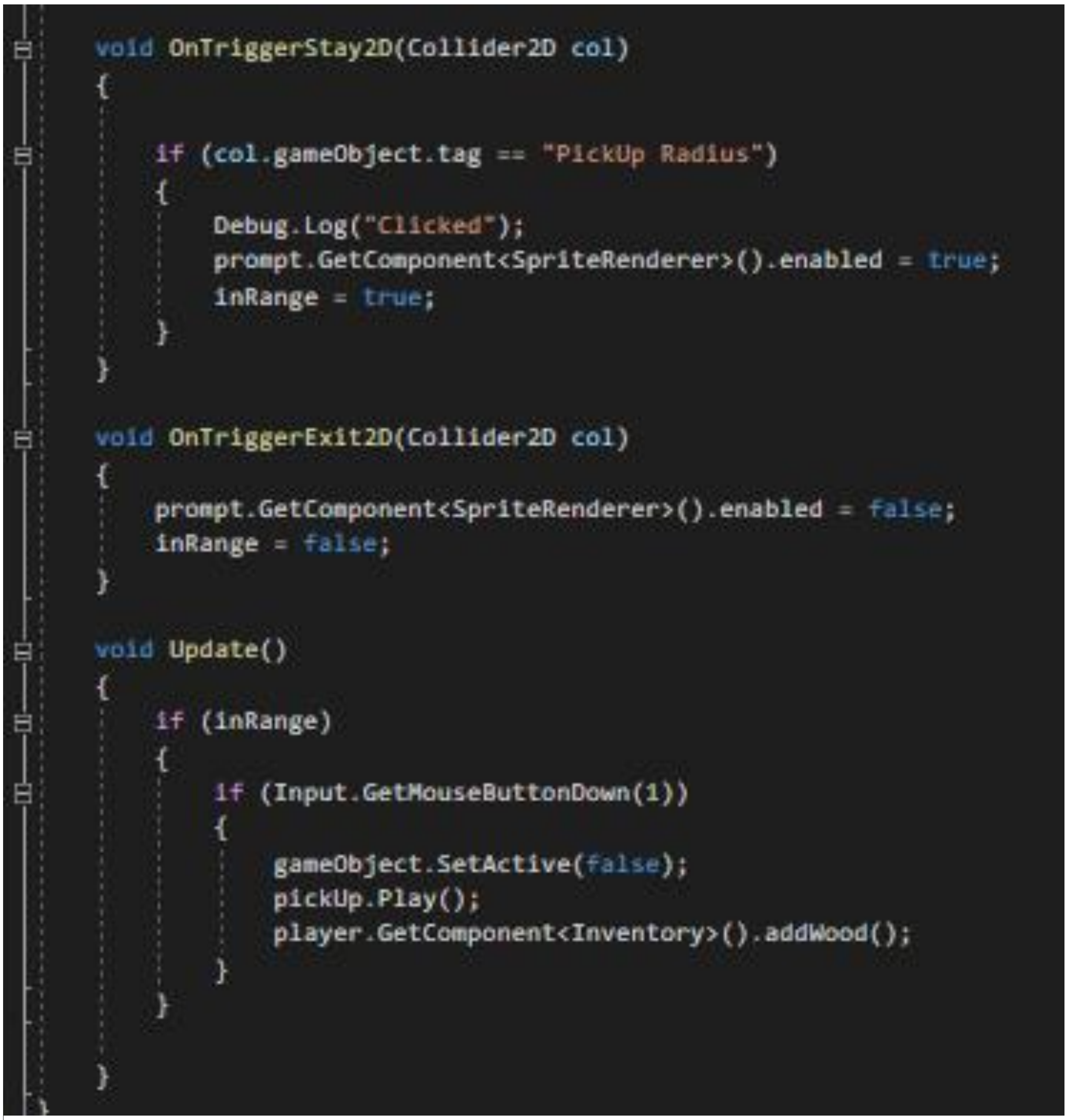
The player avatar can move in 8 directions. When doing so, the player controller will decide which animation to play, and apply a rigid body force onto the player object to create the illusion of movement (this also applies to all AI agents with movement capabilities). Having a collider component attached to the player also means that they are confined to the play-space and will collide with other objects within the scene.



```
waterVal = waterVal - 0.025f;
waterBar.fillAmount = waterVal / 1000.0f;

if (waterVal < 0)
{
    healthVal -= 0.1f;
    healthBar.fillAmount = healthVal / 1000.0f;
}

movement.x = Input.GetAxisRaw("Horizontal");
movement.y = Input.GetAxisRaw("Vertical");

anim.SetFloat("Horizontal", movement.x);
anim.SetFloat("Vertical", movement.y);
anim.SetFloat("Speed", movement.sqrMagnitude);

//Flip the animation if the Player is going left
if (Input.GetAxisRaw("Horizontal") > 0)
{
    _renderer.flipX = false;
}
else if (Input.GetAxisRaw("Horizontal") < 0)
{
    _renderer.flipX = true;
}
```

Figure 34 - Code Snippet Showing Player Vitals and Animation Conditions

With a variety of AI and items dotted throughout the level, the player has the ability to swing their weapon (through keyboard input) within range to get an effect. Resources will break and drop a collectible, and AI agents will take damage or drop food. Picking up items has been implemented using a circular trigger that encompasses an area around the player paired with a script that destroys all (obtainable) objects within this area and adds them into their inventory (details on the implementation of the inventory can be found in the Game Mechanics section (below)).



```
void OnTriggerStay2D(Collider2D col)
{

    if (col.gameObject.tag == "PickUp Radius")
    {
        Debug.Log("Clicked");
        prompt.GetComponent<SpriteRenderer>().enabled = true;
        inRange = true;
    }
}

void OnTriggerExit2D(Collider2D col)
{
    prompt.GetComponent<SpriteRenderer>().enabled = false;
    inRange = false;
}

void Update()
{
    if (inRange)
    {
        if (Input.GetMouseButtonDown(1))
        {
            gameObject.SetActive(false);
            pickUp.Play();
            player.GetComponent<Inventory>().addWood();
        }
    }

}
}
```

Figure 35 - Code Snippet to Pick Up Wood

## GAMEPLAY MECHANICS

### ITEMS

#### FOOD

Food items are acquired through harvesting specific natural resources or killing the NPC animals that roam the play-space. Vegetables are abundant but recover little health, and meat is rarer and recovers more health than vegetables. They have the added challenge however because you will need to chase / defeat the AI it is harvested from.

### INVENTORY

By using a script attached to the game UI container, we can connect integer counters to UI text elements. Then, we create another script that is attached to the collectable game object, which destroys it upon interaction and increments the integer counter. The player can view the contents of their inventory through the game UI panel on the bottom-left of the screen.

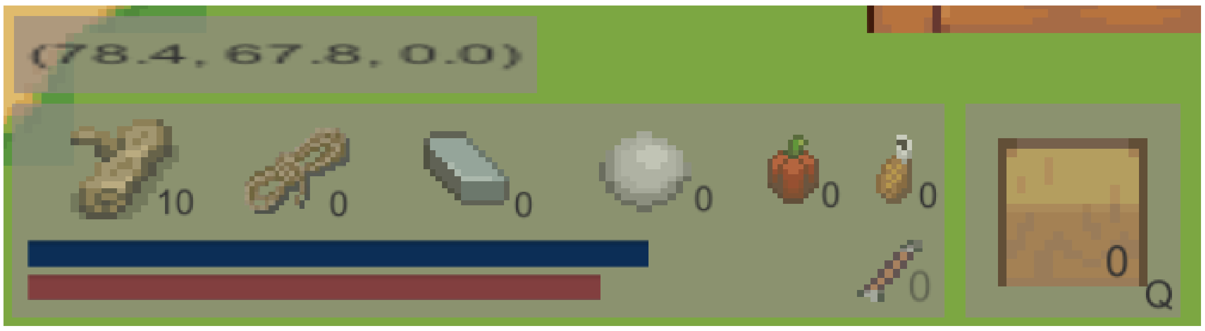


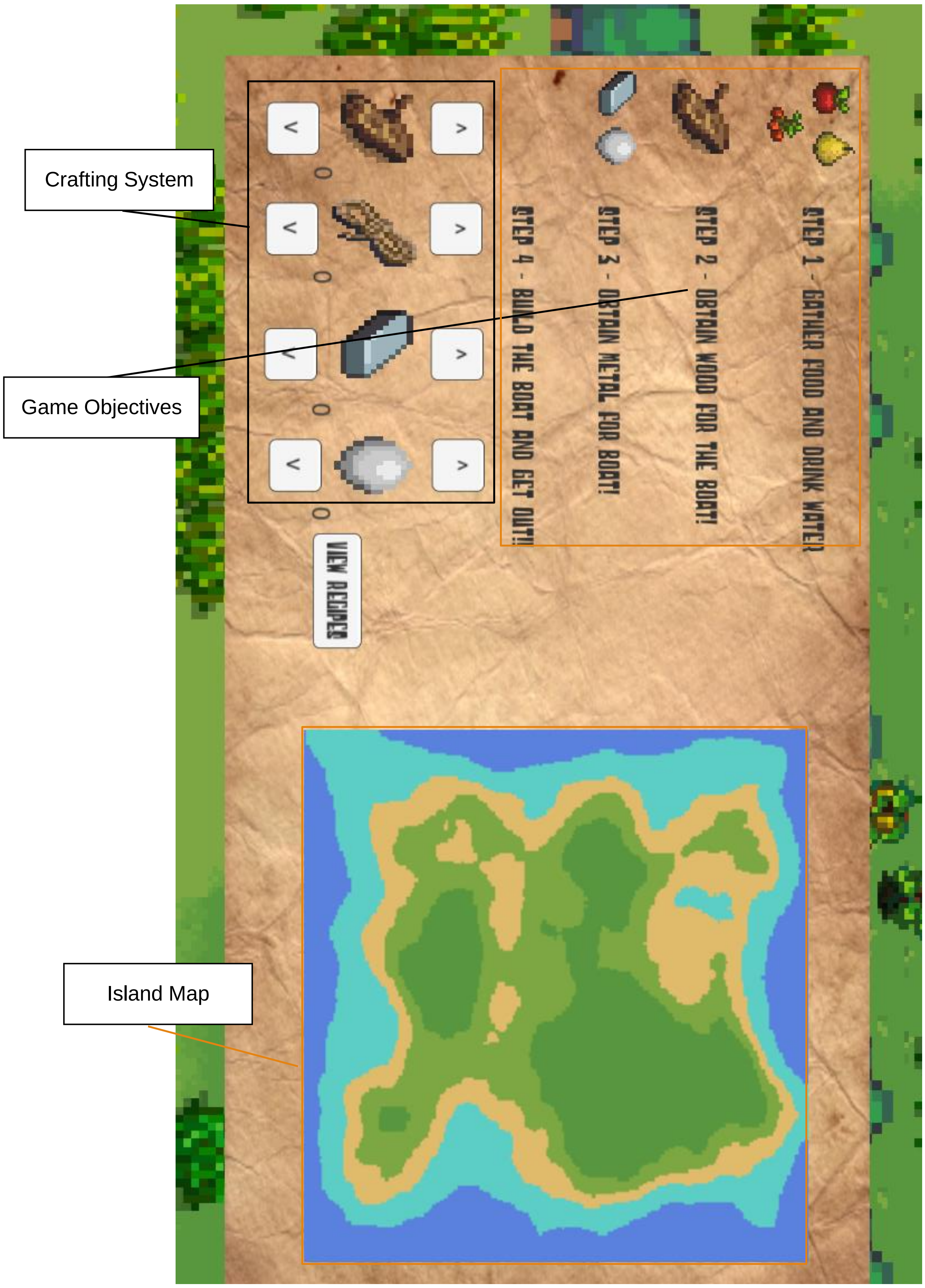
Figure 36 - hosTILE Game HUD

The figure above shows hosTILE’s UI. It is placed on the bottom right side of the player’s screen and provides the player with all the necessary information to allow them to understand the state of play or make informed decisions.

Coordinates can be utilised to return to areas such as builds, cave entrances or otherwise significant locations.

The blue bar represents the avatar’s hydration, and the red bar represents their health. The water bar will be constantly draining at a steady pace, and the health bar is only reduced upon taking damage or when the water bar runs out.

Pressing Q or E will scroll through the game mode, allowing multiple uses for the same controls.



STEP 1 - GATHER FOOD AND DRINK WATER!

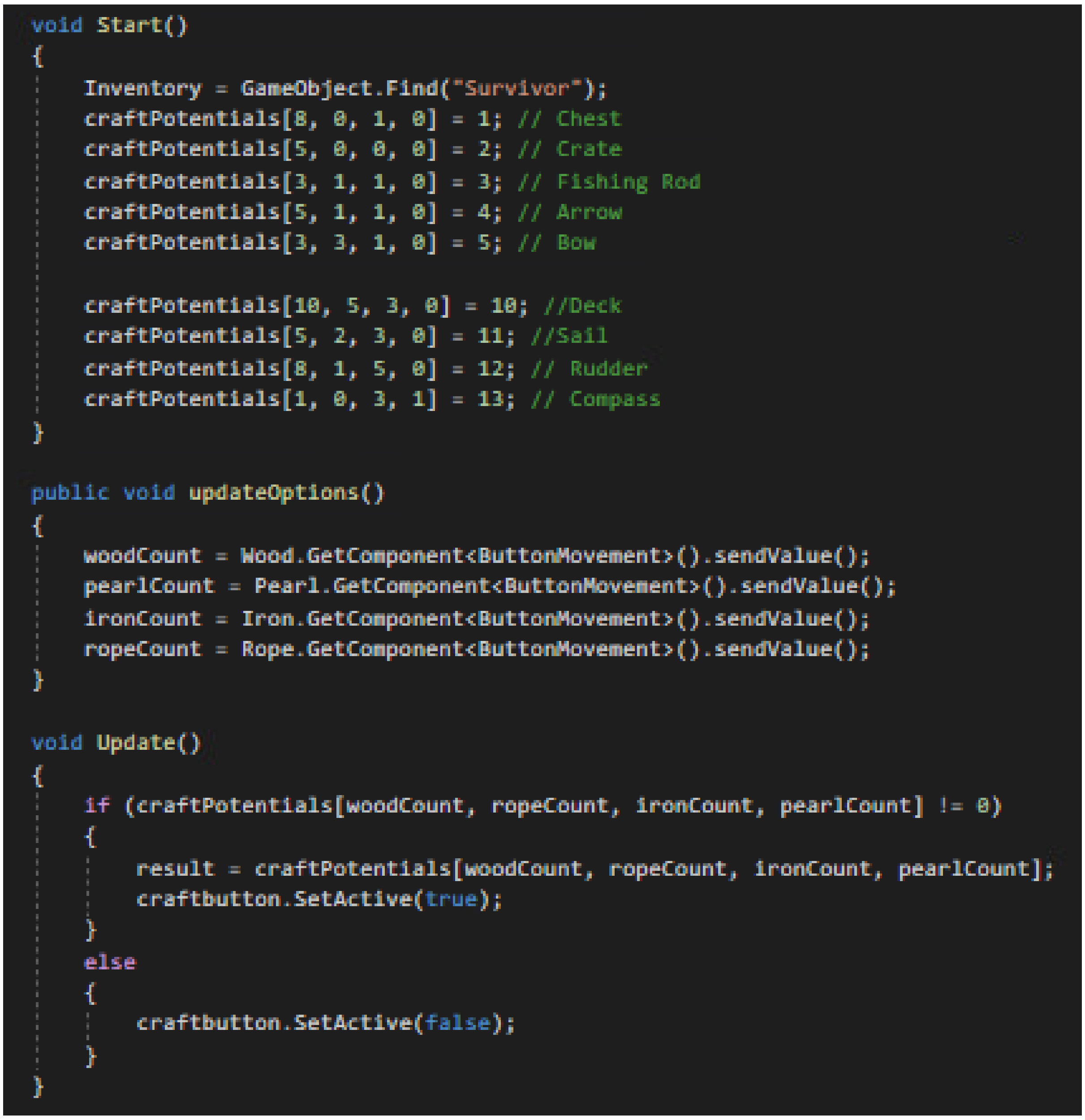
STEP 2 - OBTAIN WOOD FOR THE BOAT!

STEP 3 - OBTAIN METAL FOR BOAT!

STEP 4 - BUILD THE BOAT AND GET OUT!!

Figure 37 - In-Game Screenshot of the Inventory Menu

The challenge of creating a crafting menu was solved using 4-Dimensional integer arrays. The crafting system within the game will make use of 4 items used in different quantities to produce different results. The constituents being wood, rope, iron, and pearl. In the method devised for the game, the quantity of each respective item reflects coordinate points in the array. If the cell contains a number greater than 0, then an item can be created. The element in the array will have an integer number which represents the item to be made, so once a valid recipe has been entered, the script will loop through the numbers until there is a match, where it will subsequently instantiate said object onto the level. For example, to make a storage chest, the requirements are 8 wood, 0 pearls and 1 iron. [8, 0, 1] in the array contains the number 1, and the number 1 in the switch statement will find the storage chest game object and place it into the scene.



```
void Start()
{
    Inventory = GameObject.Find("Survivor");
    craftPotentials[8, 0, 1, 0] = 1; // Chest
    craftPotentials[5, 0, 0, 0] = 2; // Crate
    craftPotentials[3, 1, 1, 0] = 3; // Fishing Rod
    craftPotentials[5, 1, 1, 0] = 4; // Arrow
    craftPotentials[3, 3, 1, 0] = 5; // Bow

    craftPotentials[10, 5, 3, 0] = 10; //Deck
    craftPotentials[5, 2, 3, 0] = 11; //Sail
    craftPotentials[8, 1, 5, 0] = 12; // Rudder
    craftPotentials[1, 0, 3, 1] = 13; // Compass
}

public void updateOptions()
{
    woodCount = Wood.GetComponent<ButtonMovement>().sendValue();
    pearlCount = Pearl.GetComponent<ButtonMovement>().sendValue();
    ironCount = Iron.GetComponent<ButtonMovement>().sendValue();
    ropeCount = Rope.GetComponent<ButtonMovement>().sendValue();
}

void Update()
{
    if (craftPotentials[woodCount, ropeCount, ironCount, pearlCount] != 0)
    {
        result = craftPotentials[woodCount, ropeCount, ironCount, pearlCount];
        craftbutton.SetActive(true);
    }
    else
    {
        craftbutton.SetActive(false);
    }
}
```

Figure 38 - Code Snippet for Crafting

Although this approach appears to be costly from an outward perspective, there is no need to cycle through every single element. Changing an item's quantity causes the script to reference the coordinate directly, and further code blocks are only executed when a valid recipe is entered.

The below table shows the items that can be crafted in the game, alongside the recipe required to make them.

## CRAFTING MATERIALS

Crafting materials have no use until partnered with other materials. The result of crafting can be a second stage crafting material, or a tool. If the player wishes to complete the game, they will need to accumulate resources and craft the items necessary to build the boat. Within the game, there are four crafting resources:

| Resource | Found |
|---|---|
| Wood | Trees |
| Iron | Rocks, Mineral Deposits |
| Rope | Plants |
| Pearls | Clams |

Figure 39 - Table Showing Crafting Resources and Locations

| Item | Use | Recipe (Wood, Rope, Iron, Pearl) |
|---|---|---|
| Bow | Long Range Attacks | 3, 3, 1, 0 |
| Arrow | Bow Ammunition | 5, 1, 1, 0 |
| Wall | Building Item | 5, 0, 0, 0 |
| Looking Glass | Zoom In and Out | 0, 0, 3, 2 |
| Deck | Boat Part | 10, 5, 3, 0 |
| Rudder | Boat Part | 8, 1, 5, 0 |
| Sail | Boat Part | 5, 2, 3, 0 |
| Compass | Boat Part | 1, 0, 3, 1 |

Figure 40 - Table Showing Craftable Items and Their Uses

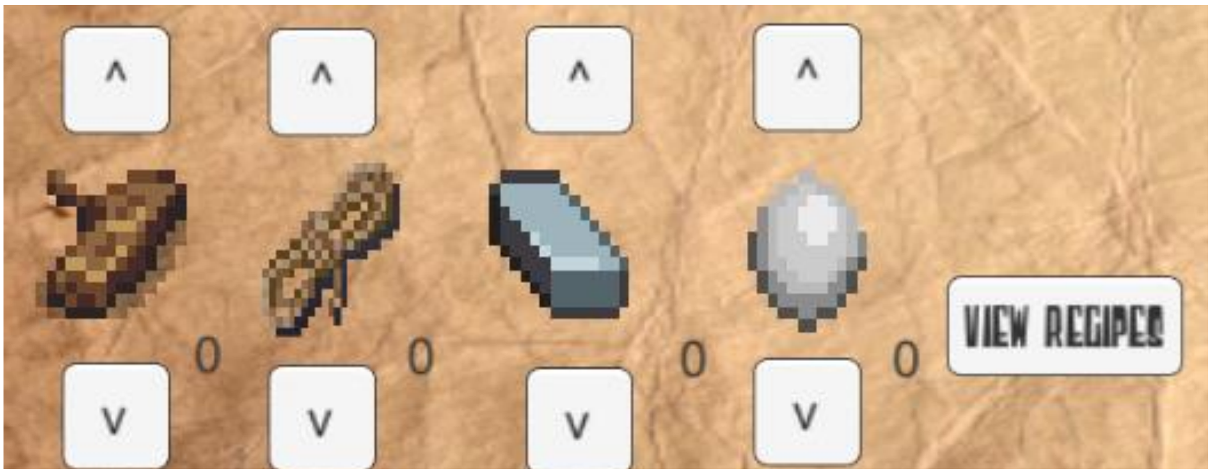
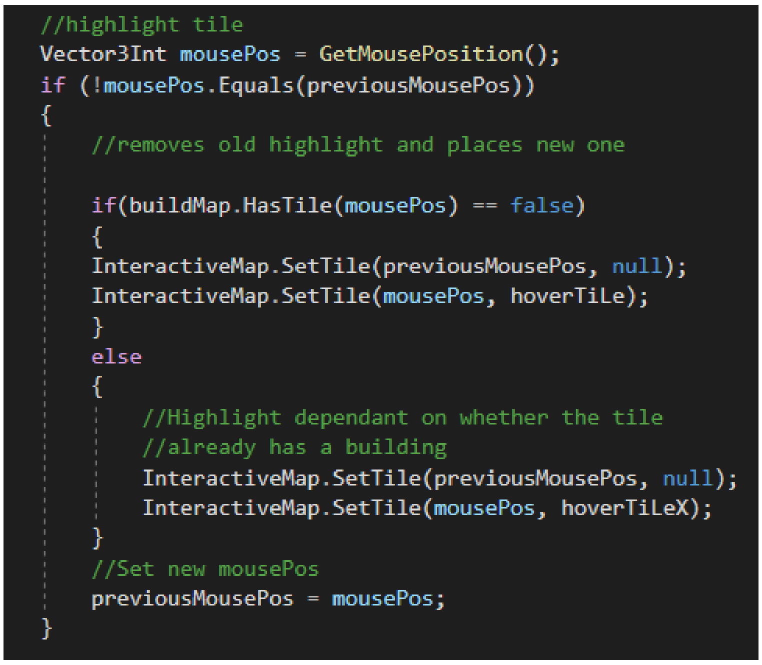


Figure 41 - In-Game Look at the Crafting Menu

## BUILDING

With such an open world, the game invites creativity. Adding a building mechanic allows the player to create their own buildings, form boundaries to keep enemies away their territory. Pressing Q will change the game mode and permit the player to place tiles wherever they click on screen. Using this mechanic however means that the player cannot attack, mine, or use consumables, leaving them vulnerable. The items that can be used in the build mode are crafted. The building system works by placing tiles where the player clicks. These tiles are on their own grid as to not affect the rule tiles of the overworld, and the selector is also on a separate grid. The selector is removed from the tile once the mouse goes out of the cell and is then instantiated onto the new cell. If the selector is being held above an already taken slot, it will change to a red cross and not allow the player to use a block.



```
//highlight tile
Vector3Int mousePos = GetMousePosition();
if (!mousePos.Equals(previousMousePos))
{
    //removes old highlight and places new one

    if(buildMap.HasTile(mousePos) == false)
    {
    InteractiveMap.SetTile(previousMousePos, null);
    InteractiveMap.SetTile(mousePos, hoverTiLe);
    }
    else
    {
        //Highlight dependant on whether the tile
        //already has a building
        InteractiveMap.SetTile(previousMousePos, null);
        InteractiveMap.SetTile(mousePos, hoverTiLeX);
    }
    //Set new mousePos
    previousMousePos = mousePos;
}
```

Figure 42 - Code Snippet for Highlighting Tile Cells

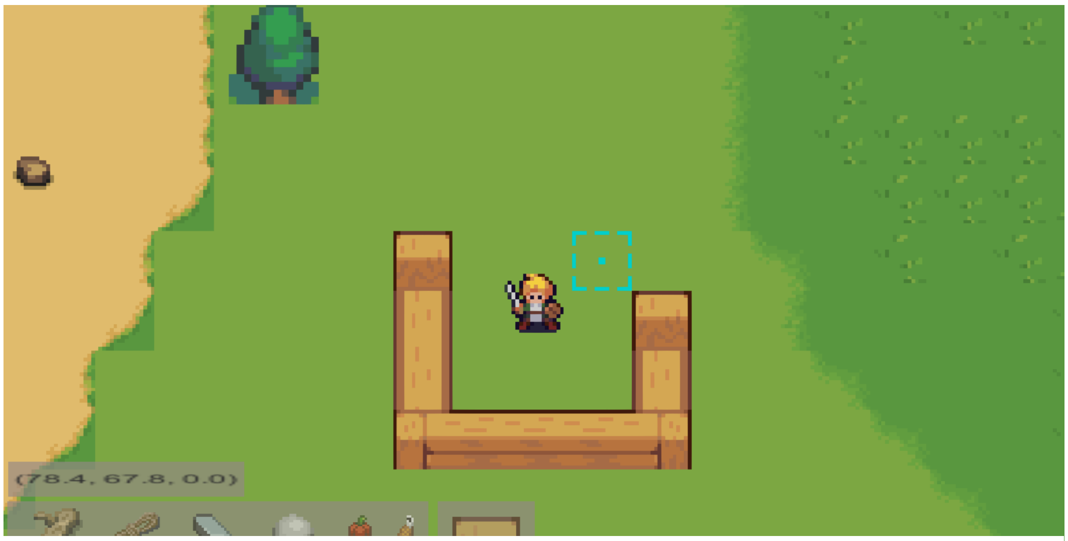


Figure 43 - Screenshot of Build Mechanic

## ARTIFICIAL INTELLIGENCE

As mentioned at the beginning of this paper, the game will be populated with artificially intelligent units in the form of animals and monsters. These units will have similar functionality to one another; however, each individual class will have its own unique behavioural pattern to add interactive diversity to the game's ecosystem.

| Species | Traits | Aggression |
|---|---|---|
| Crab | Sideways movement, no response | Passive |
| Ducks | Four directional movement, no response, herbivore | Passive |
| Rabbits | Four directional movement, flight response, herbivore | Passive |
| Slime | Four directional movement, fight response, predator | Aggressive |

**Figure 44 - Table to Show AI behaviours**

### ARTIFICIAL INTELLIGENCE MODEL

Typical artificial intelligence models are comprised of three sections: strategy, decision making, and movement (Jaokar, 2019). Each section tackles specific problems for the AI object, and when assembled, result in the action taken by the agent. These sections are influenced by the world of the game, for example, if a passive, flight response AI is in close proximity to a threat (i.e., the player or a predator), the decision making section will determine the best direction to attempt an escape. This assessment will then persuade the movement unit, which will apply the correct physics and animation to perform motion on-screen away from the threat. Figure 45 shows this model. Within the context of the game, strategy AI is not relevant, as it refers to the tactics used by board game agents and the coordination of teams of units. The units in the game will be individual and react to their own scenarios.

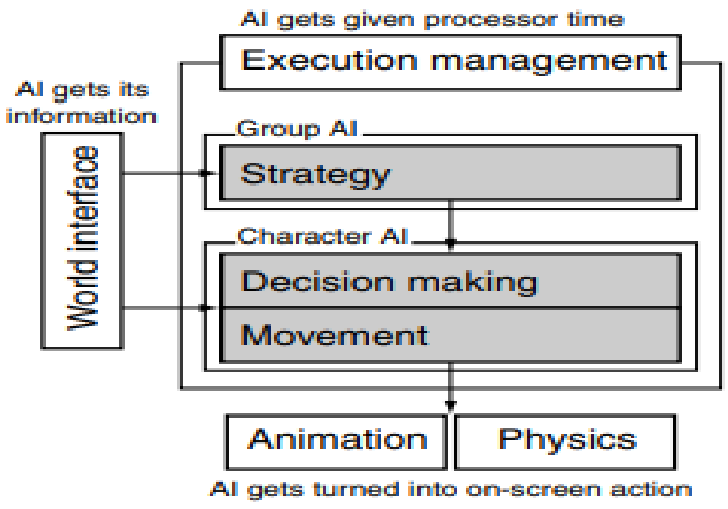


**Figure 45 - The AI Model (Millington and Funge, 2009)**

## ALGORITHMS AND REPRESENTATIONS

The Algorithms are the code that goes behind the Unity game objects that performs the relevant calculations and response, however this alone is not enough to create fluid, seemingly intelligent AI. There needs to be a context to the game world as well so the unit can perceive its surroundings. Going back to our previous example of the passive, flight response AI, we need to be able to identify the threats as they enter the animal's field of view. A polygon collider set as a trigger and being placed in front of the AI acts like their field of view. If the trigger is prompted, the code will retrieve the tag of the incoming collider and begin the motions of reaction if the tag corresponds to a threat. This way the unit reacts only to the appropriate collider intersections. The game object can be marked as such, and the algorithm can obtain its transform data for use with its escape calculations.

## IMPLEMENTATION OF AI

It is important to note that there are two types of AI: deterministic and non-deterministic. Deterministic AI refers to those whose actions can be exactly predicted through the environment, excluding the participation of randomness (Robins, 2020). Non-deterministic (otherwise referred to as stochastic) AI can provide multiple outcomes based on the same input, meaning that it is not possible to accurately predict an agent's response (What is a Non-Deterministic Algorithm? - Definition from Techopedia, 2019). The AI used within hosTILE will follow the design of basic non-deterministic AI. The stochastic behaviours are derived from the usage of many randomly generated numbers used to set the target positions, emulating a real animal roaming.

### PASSIVE AGENTS

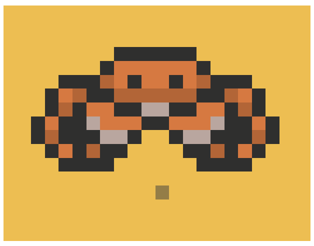
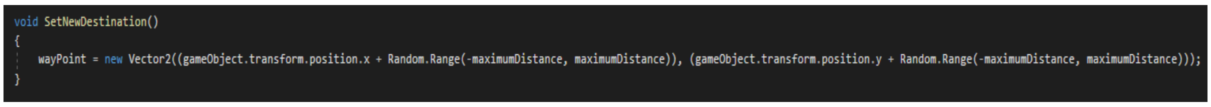


Figure 46 - Passive Crab Agent

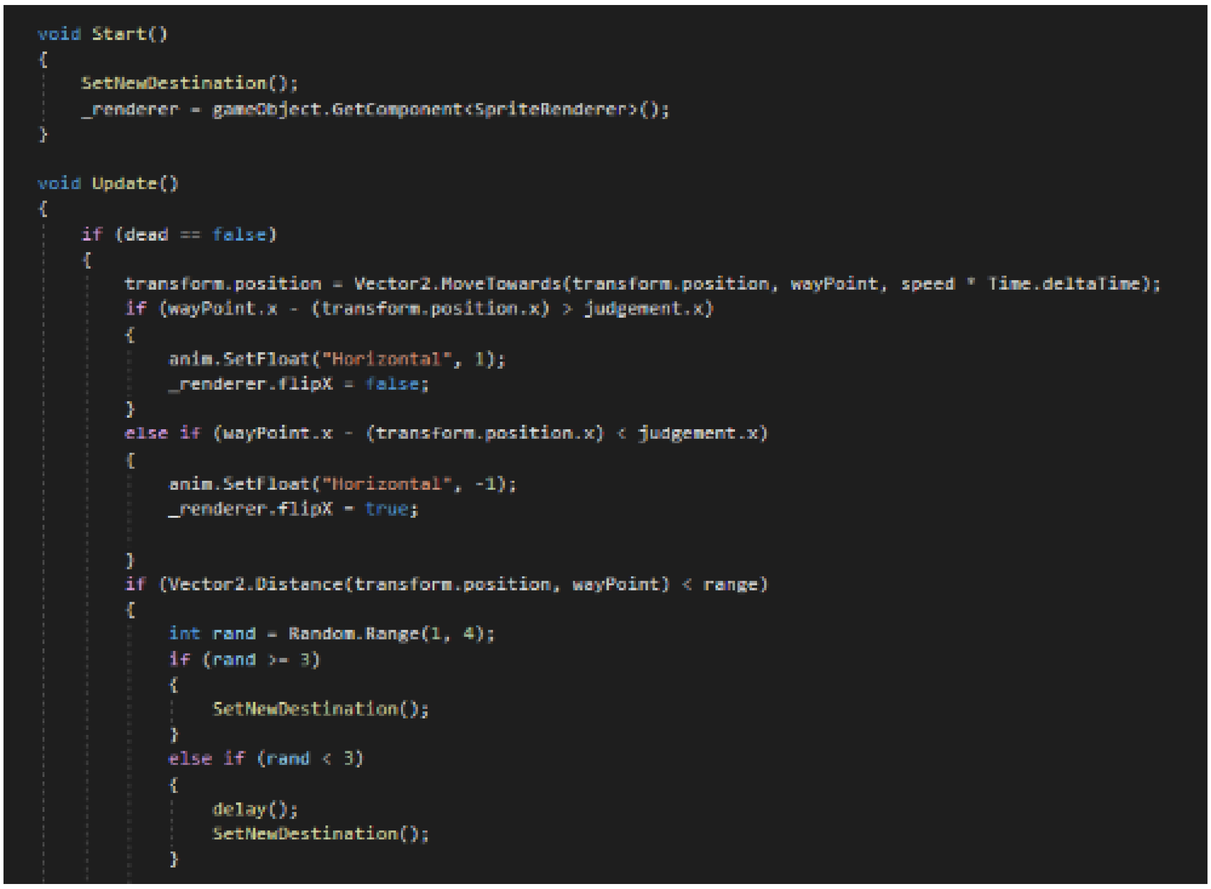
Passive Agents within the game do not come into conflict with their neighbouring NPCs. Some do not react to hostile scenarios, and aimlessly wander their ecosystem, whilst others actively avoid threats by moving directly away until the danger is out of their view. Herbivorous AI will target and approach vegetables that the player does not pick up, allowing traps to be set. If these animals are on the drop for long enough, they will eat it.

## AGGRESSIVE AGENTS



```
void SetNewDestination()
{
    wayPoint = new Vector2((gameObject.transform.position.x + Random.Range(-maximumDistance, maximumDistance)), (gameObject.transform.position.y + Random.Range(-maximumDistance, maximumDistance)));
}
```

Figure 48 - Code Snippet Showing How the Random Position was Calculated



```
void Start()
{
    SetNewDestination();
    _renderer = gameObject.GetComponent<SpriteRenderer>();
}

void Update()
{
    if (dead == false)
    {
        transform.position = Vector2.MoveTowards(transform.position, wayPoint, speed * Time.deltaTime);
        if (wayPoint.x - (transform.position.x) > judgement.x)
        {
            anim.SetFloat("Horizontal", 1);
            _renderer.flipX = false;
        }
        else if (wayPoint.x - (transform.position.x) < judgement.x)
        {
            anim.SetFloat("Horizontal", -1);
            _renderer.flipX = true;

        }
        if (Vector2.Distance(transform.position, wayPoint) < range)
        {
            int rand = Random.Range(1, 4);
            if (rand >= 3)
            {
                SetNewDestination();
            }
            else if (rand < 3)
            {
                delay();
                SetNewDestination();
            }
```

Figure 47 - Code Snippet Showing Agent Movement to a New Position

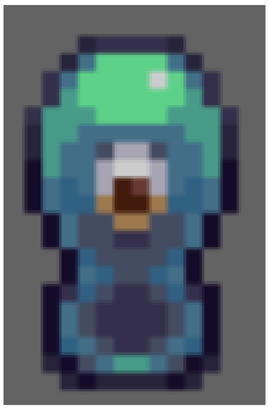
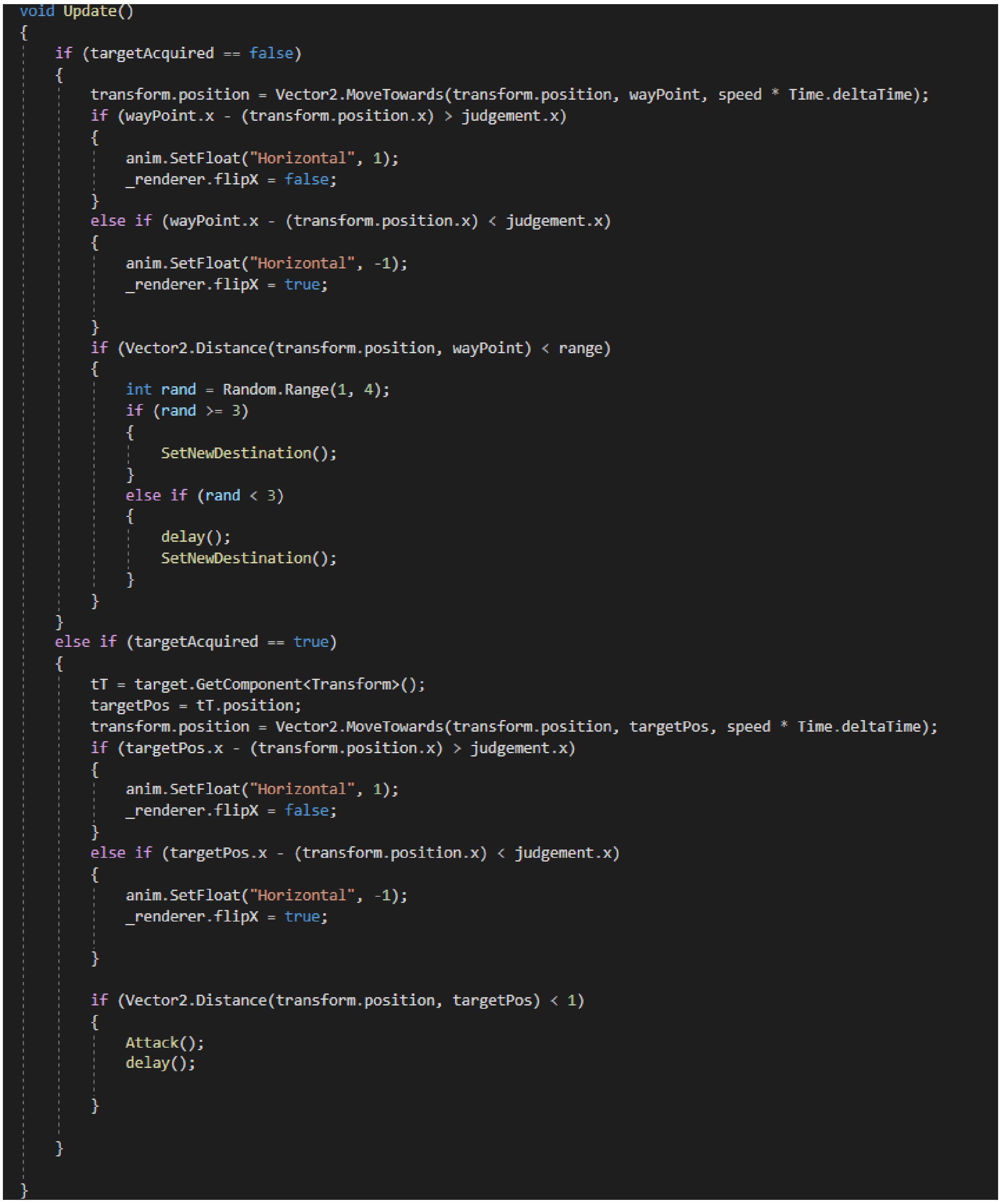


Figure 49 - Slime Enemy

Aggressive agents will energetically move around their environment in search of prey; be it the other wildlife or the player. They lock onto said target using a circle trigger that encompasses their sprite model. The first object to enter this circle will have their transform positions taken and they will make a move for that location until they either reach it or the target escapes their view.



```
void Update()
{
    if (targetAcquired == false)
    {
        transform.position = Vector2.MoveTowards(transform.position, wayPoint, speed * Time.deltaTime);
        if (wayPoint.x - (transform.position.x) > judgement.x)
        {
            anim.SetFloat("Horizontal", 1);
            _renderer.flipX = false;
        }
        else if (wayPoint.x - (transform.position.x) < judgement.x)
        {
            anim.SetFloat("Horizontal", -1);
            _renderer.flipX = true;

        }
        if (Vector2.Distance(transform.position, wayPoint) < range)
        {
            int rand = Random.Range(1, 4);
            if (rand >= 3)
            {
                SetNewDestination();
            }
            else if (rand < 3)
            {
                delay();
                SetNewDestination();
            }
        }
    }
    else if (targetAcquired == true)
    {
        tT = target.GetComponent<Transform>();
        targetPos = tT.position;
        transform.position = Vector2.MoveTowards(transform.position, targetPos, speed * Time.deltaTime);
        if (targetPos.x - (transform.position.x) > judgement.x)
        {
            anim.SetFloat("Horizontal", 1);
            _renderer.flipX = false;
        }
        else if (targetPos.x - (transform.position.x) < judgement.x)
        {
            anim.SetFloat("Horizontal", -1);
            _renderer.flipX = true;

        }

        if (Vector2.Distance(transform.position, targetPos) < 1)
        {
            Attack();
            delay();

        }

    }

}
```

Figure 50 - Code Snippet of Predator Movement

## MENUS

When first booting up a game the first thing the player is introduced to is the main menu. It is a key constituent of the game that allows the player to do a variety of things such as play the game, change settings, or exit.

This menu scene was created to fit that aesthetic of the project at hand. The below image (Figure 51) is the design for the main menu of the game. It shows a desert island with animated water washing up onto the beach. The left side of the island has the three options ready for selection, whilst on the right we have some rocks and a raised bit of land with some greenery. Although the layout of the buttons is slightly different to those shown in the case study, the menu provides the player with insight into the design and aesthetic of the game as well as remaining simple and readable.

The settings button expands to show video, audio, and controls options which, themselves open to display a variety of amendable selections. Making a selection in these menus will make a change to the game when play is pressed, such as lowering the volume of particular sound groups, entering full screen and changing the image resolutions. There is a tutorial button which will lead the player to a smaller scale island, generated using the developed PCG tool as well as developer additions such as noticeboards that grant information about the gameplay and how the mechanics work.

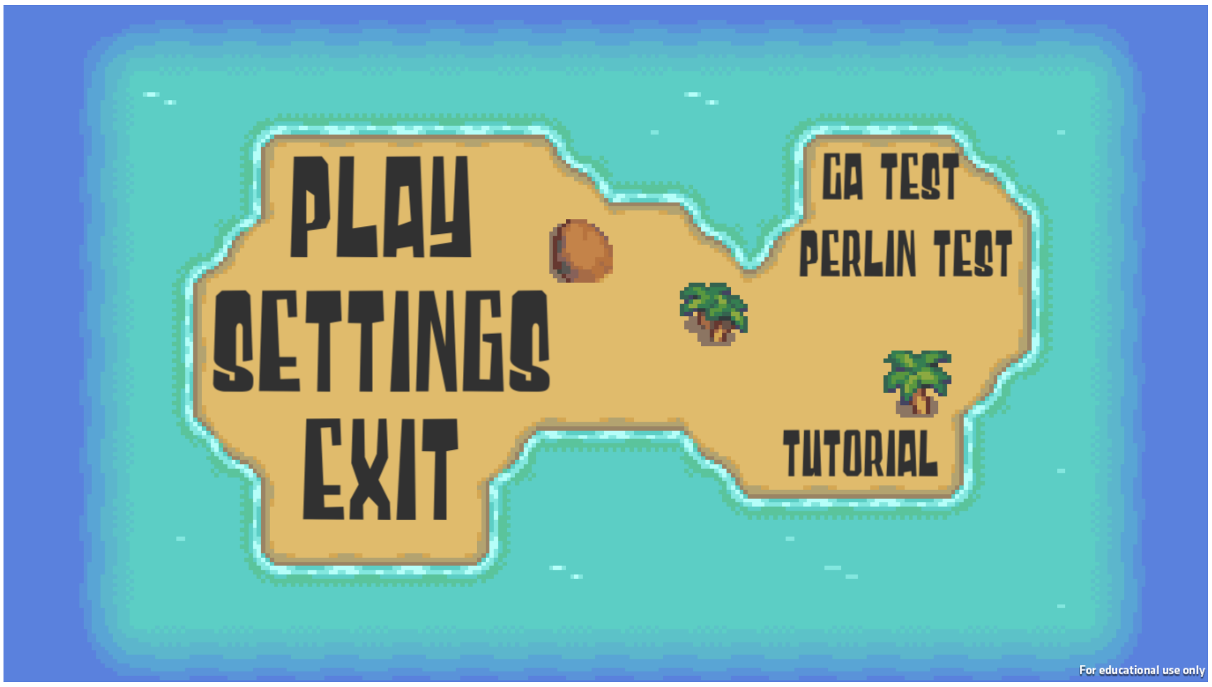


Figure 51 - hosTILE Main Menu

## SOUNDTRACK

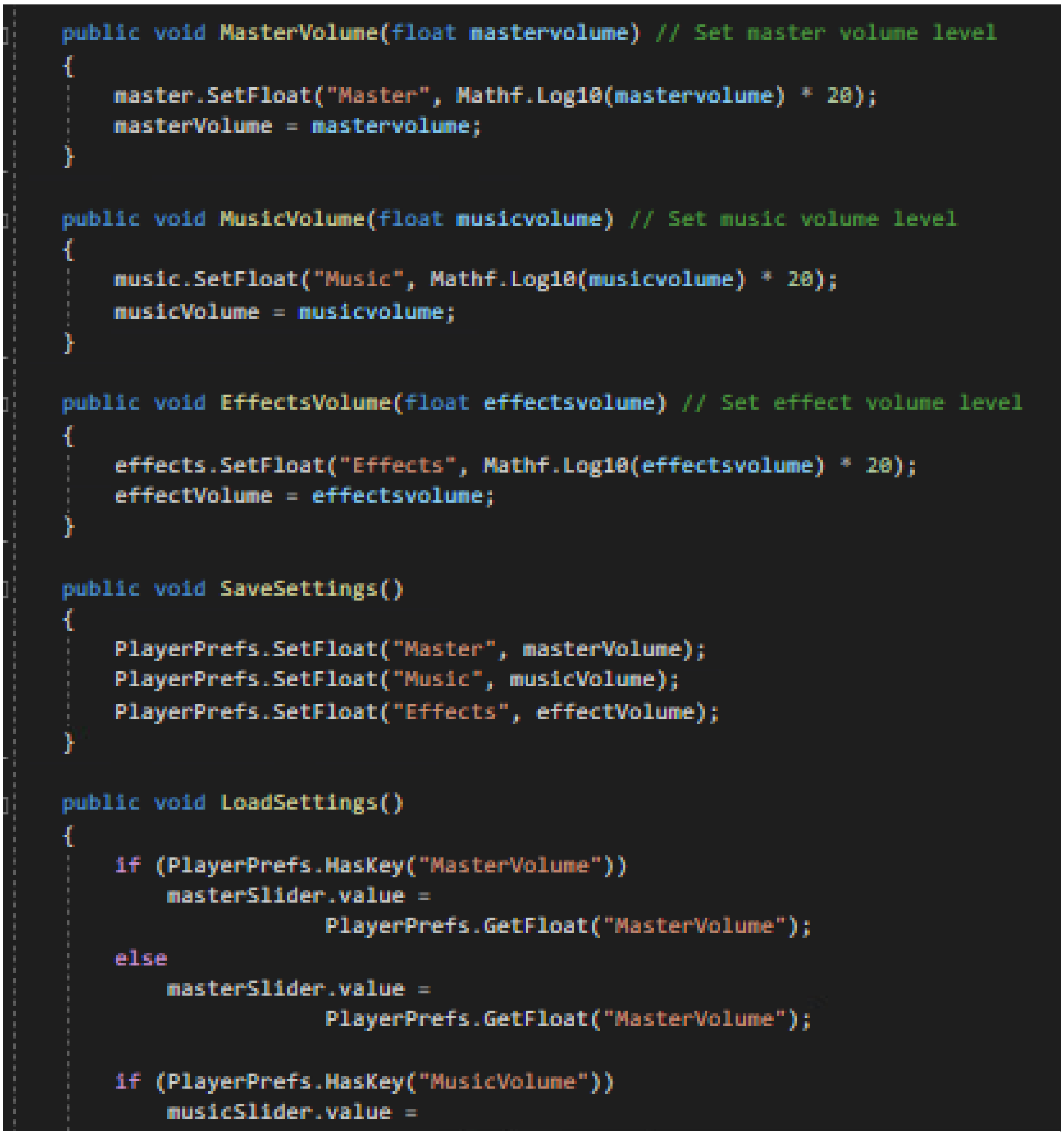
### MUSIC

To fit the writer’s vision, the music for hostile was created especially for the game. Taking heavy inspiration nature documentaries and other nature themed medias, the soundtrack makes use of ambient sounds that suitably accompanies the gameplay without distracting the player from the game itself. To add a sense of loneliness the music is sombre, to reflect the player’s avatar being stranded. The order of the tracks will be randomised, so the player is greeted with different tracks upon beginning each game.

This is a music player program. It is attached to an empty object and is used to play the soundtrack in a shuffled order. Within the editor, the created tracks can be placed into the array. Once an audio clip has been selected by the script, it is placed into the audio source component of the music player, which plays on awake. Now during runtime, the music tracks will be played randomly.

## SOUND EFFECTS AND AUDIO GROUPS

With the general aesthetic of the game, the obtained sound effect clips are short and to the point. Their job is to add sensation and player feedback when an action is taken. For example, hitting a rock will play a brief “clunk” to emulate the real world event. HUD and UI interactions also produce feedback as a way of confirming selections. All sounds were sourced from an online library of royalty free audio clips. These assets are credited in the appendix. All games have a division between music, sound effects, and cinematic noises, which can be amended in the sounds page in both the pause and main menu screens. This allows for these specific sound groups to be amplified or reduced to satisfy the player’s preferences. This is done through creating audio groups and allowing a script to amend the volumes.



```
public void MasterVolume(float mastervolume) // Set master volume level
{
    master.SetFloat("Master", Mathf.Log10(mastervolume) * 20);
    masterVolume = mastervolume;
}

public void MusicVolume(float musicvolume) // Set music volume level
{
    music.SetFloat("Music", Mathf.Log10(musicvolume) * 20);
    musicVolume = musicvolume;
}

public void EffectsVolume(float effectsvolume) // Set effect volume level
{
    effects.SetFloat("Effects", Mathf.Log10(effectsvolume) * 20);
    effectVolume = effectsvolume;
}

public void SaveSettings()
{
    PlayerPrefs.SetFloat("Master", masterVolume);
    PlayerPrefs.SetFloat("Music", musicVolume);
    PlayerPrefs.SetFloat("Effects", effectVolume);
}

public void LoadSettings()
{
    if (PlayerPrefs.HasKey("MasterVolume"))
        masterSlider.value =
                    PlayerPrefs.GetFloat("MasterVolume");
    else
        masterSlider.value =
                    PlayerPrefs.GetFloat("MasterVolume");

    if (PlayerPrefs.HasKey("MusicVolume"))
        musicSlider.value =
```

Figure 52 - Code Snippet for Audio Settings

## PROCEDURAL DEMONSTRATIONS

To show how each variable effects the outcome of a generated level, two scenes have been made that allows the user to input their own values to demonstrate the procedural techniques present within the portfolio. By using UI sliders, the user can explore a variation of settings to better understand these techniques. Some values are within boundaries, because straying too far from these values will have a negative effect on the outcome.

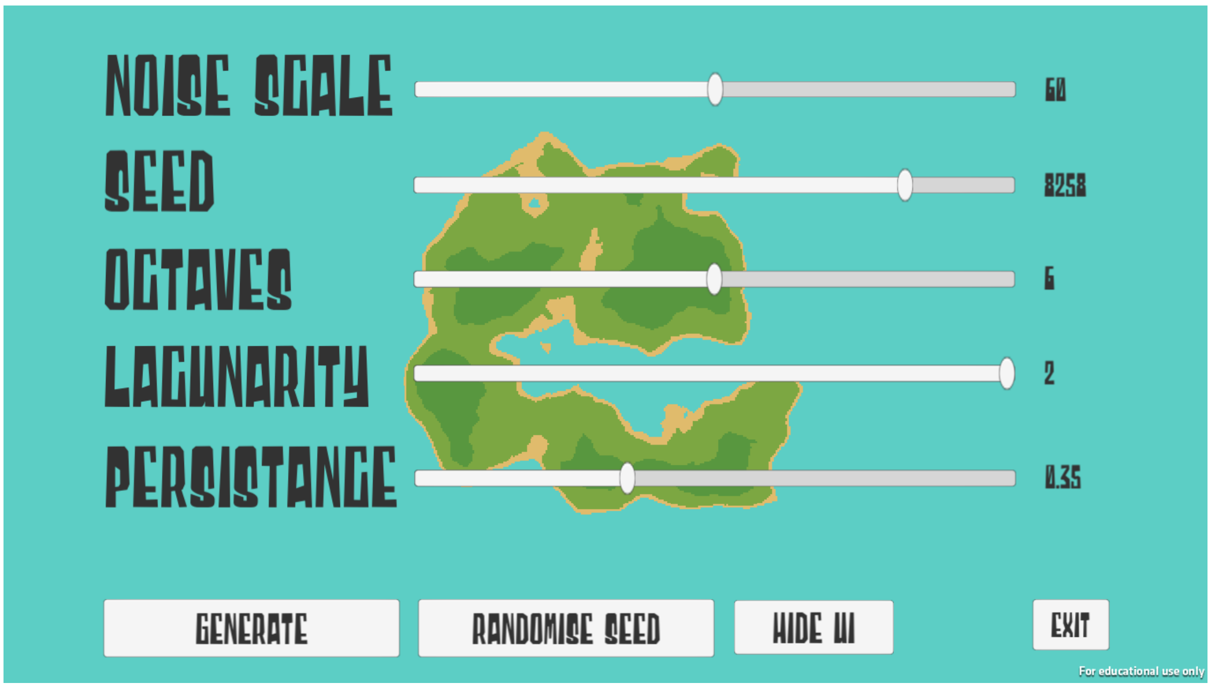

NOISE SCALE 60
SEED 8258
OCTAVES 6
LACUNARITY 2
PERSISTANCE 0.35

Figure 53 - Perlin Demo Scene

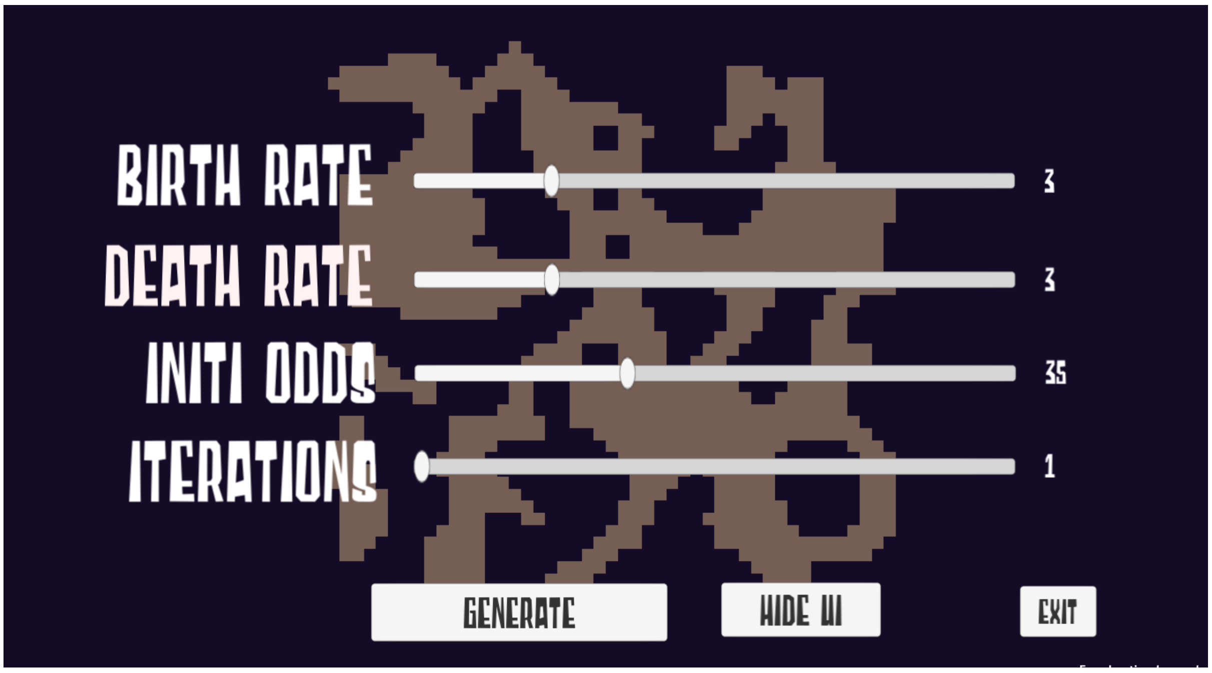

BIRTH RATE 3
DEATH RATE 3
INITI ODDS 35
ITERATIONS 1

Figure 54 - Cellular Automata Demo Scene

## CRITICAL REFLECTIONS

This section encompasses the writer's and 3rd party's judgement on the quality of work produced throughout this paper. It will be used as a means of determining the project's strengths and weaknesses, as well as a feed-forward tool for the writer regarding future endeavours in similar fields of game development.

### TESTING

A build of the game was sent to work colleagues as well as friends of the writer. These participants have a range of experience with games, and therefore notice different matters that they bring forward in the feedback. A questionnaire was formulated asking specific questions about the game and the user's experience. The following sections encompass the third-party user feedback to constructively measure the success of the game.

#### FEEDBACK

Overall, the feedback received was positive. Responses enjoyed or were content with the gameplay and music. Many participants also noted that the aesthetic was a perfect choice for the genre and style of the game and enjoyed the aspect of freedom of exploration. This is an indicator that at this stage of development the game was on track to meet the writer's original vision. On top of this, the gameplay mechanics have been commended for being fun and the controls were easy to grasp by all users.

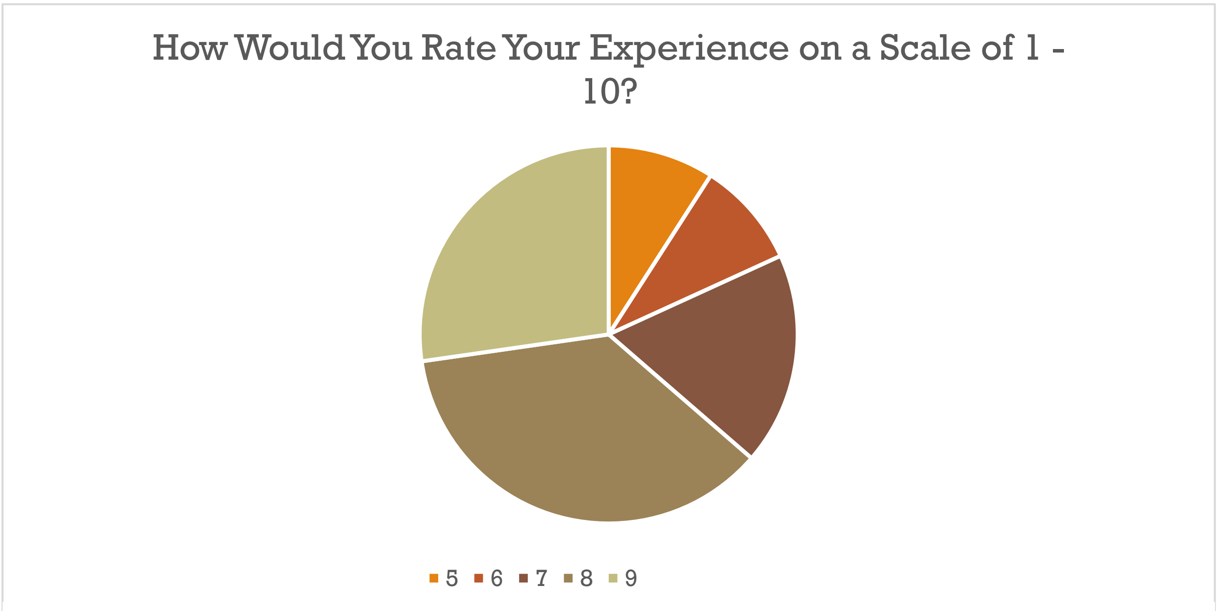


Figure 55 - Chart to Show Audience Reception for the Alpha

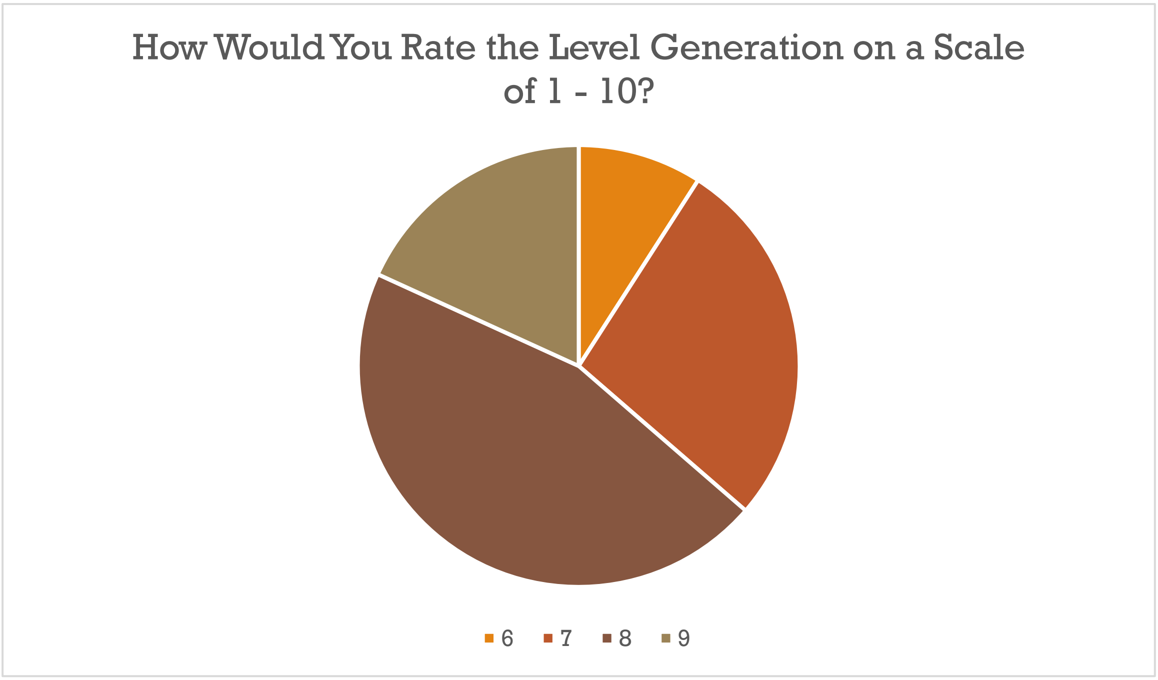


Figure 56 - Chart to Show Audience Reception of the Level Generation

Regarding the level generation side, responses indicated that while the environments produced were fun to explore and play on, there was a distinct lack of diversity in the biomes. This has presented the conclusion that adding more mechanics to interact with the environment were needed. This led to the inclusion of treasure chests that could be uncovered on the beaches. It was clear according to the replies that the game had a similar look and feel to Minecraft and Terraria, which is a positive indication that the game has a solid structure and game flow.

Asking the participants about the improvements they wished to see, many brought up bugs and glitches that they came across through their experience. These have been considered and rectified for the final product. In addition, players wished to have a bit more support with how the game worked, for example what the resources are for, as well as a clearer indication of the health and water levels ("Explanation what some of the materials I was picking up were for. Numbers on health bars to indicate exactly how much of each thing I had left"). To improve these, float values have been added to the bars on the UI to better inform the player on the state of the avatar.

## IMPROVEMENTS

Graphical breaks are common in this terrain generation model due to the sprite sheet used for the landscapes. Single divot pieces were not included, and so a single tile of any type reverted to its default sprite image as set up in the rule tiles tool. The rarity of these events has been reduced by adjusting the noise, lacunarity and persistence of the level, however it is not guaranteed that this will entirely fix the issue. This issue is only prominent in this project because the tile set utilised does not have the correct tiles to fit into the grid cell.

### OUTCOMES

Through the work put into this project, a fully finished survival game has been created. The levels are completely procedural without any human intervention and provides 20-25 minutes of play time.

## EVALUATION

### LIMITATIONS

The largest limitation for this project was having access to university computers. Due to the nature of Unity projects and the computing complexity of both cellular automata and Perlin Noise meant that my computer was not a reliable machine to work on. Connecting remotely to a campus computer worked to an extent, but the latency between action and reaction meant that the rate of work was significantly slowed when off-campus. On top of this, audio and video feeds were very slow, and as such testing the game between builds was made difficult.

As stated previously, the graphics were sourced from a third party (acknowledged in the appendix) and so I had no part in its creation. I believe that the aesthetic of said graphics were well-suited to the game, but it was missing a few tiles that would make the levels produced look smoother. Access to more complex tile sets would result in the removal of these graphical errors.

### EDUCATIONAL TAKE-AWAYS

This project has encompassed all aspects of the development cycle of a small-scale indie game. The processes tackled through each milestone has stretched the writer and made them read into a wide range of areas so that the quality and performance of the game is as clean as possible. PCG are huge topics in today’s game development scene, and the research and implementation of such allows the writer to recreate and even extend these practices in future ventures.

### FURTHER IMPROVEMENTS

The most striking area that I believe requires better implementation is the resource allocation algorithm. By using random numbers to determine when and where an object is placed means that the one-off items (i.e the cave entrance) are more likely to be instantiated in the early coordinates of the grid. The use of nested loops to sequentially move from one cell to another means that it always begins at the bottom left and finishes at the bottom right. These single objects are therefore more likely to spawn at the beginning due to the bias of probability of their inclusion, allowing for reliable predictions as to their whereabouts. I believe that going down the route of using ray casting for these objects would be more suited to better laying down these pieces as we can send a singular ray out to a completely random position on the level, check if it is a valid location, then placing it down. Not only would this aid in their distribution, but computationally it would save memory as it is not iteratively determining whether each cell is legal for use or not.

For upcoming projects, I wish to better develop my time-handling skills. The project was a large undertaking, and it would have been a better idea to focus solely on the PCG side instead of creating a full-fledged game. Management of time is something that I have enhanced during my time acting on this dissertation, and I have learned to not take on too much when given a time frame to complete a task.

## FINAL COMMENTS

This project is not without flaw, though I am extremely happy with the outcome. Perlin noise has always taken my interest, so researching and implementing on of its many uses has been a fulfilling journey. The same goes for cellular automata. Procedural generation as a field of research is such a powerful tool for games developers that educating myself and getting first-hand experience with it opens many doors in my future developments. As a developer with sparse artistic knowledge and experience, PCG is a shortcut to produce large levels without the need to create them by hand. On top of this, working solo on a project of this magnitude has boosted my confidence in areas that I previously felt I was no experienced in.

Word Count: 9910 – (1254 + 538) = 8210 words

## SCREENSHOTS

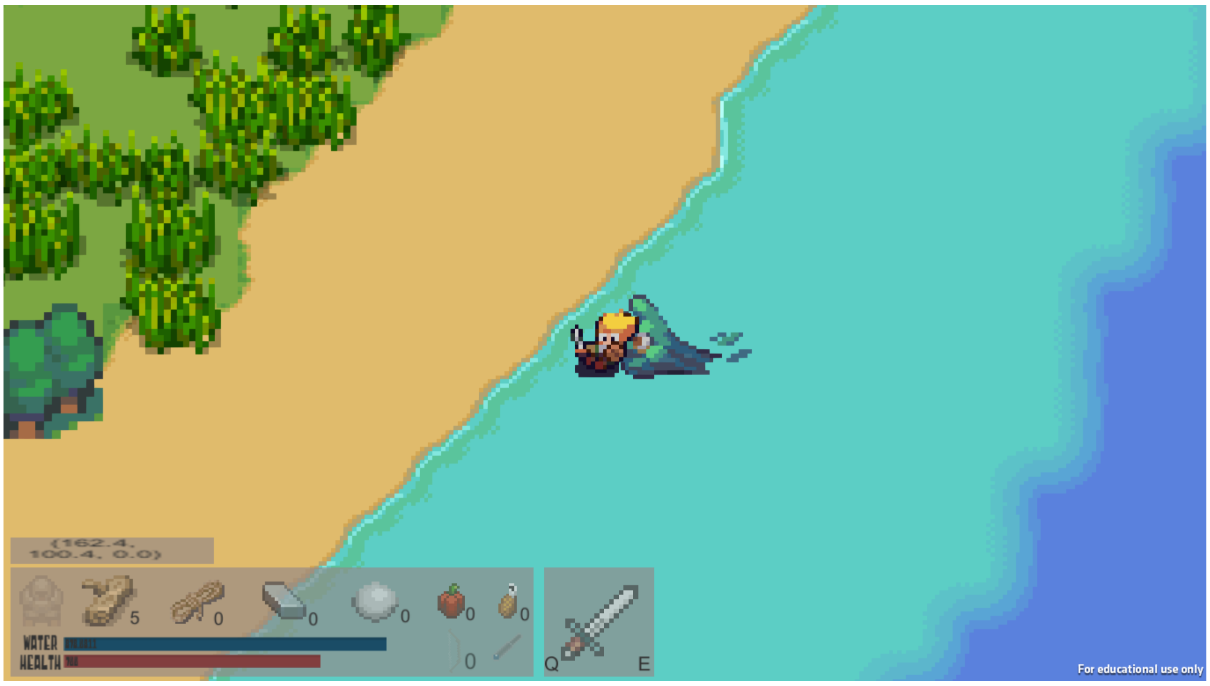


Figure 57 - Screenshot of hosTILE Gameplay

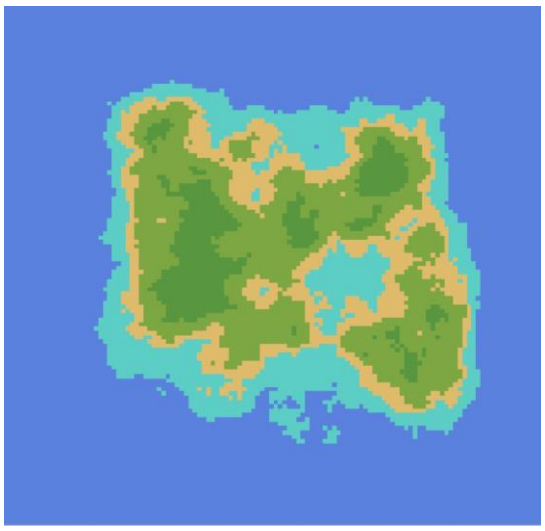


Figure 59 - Perlin Island Example 1

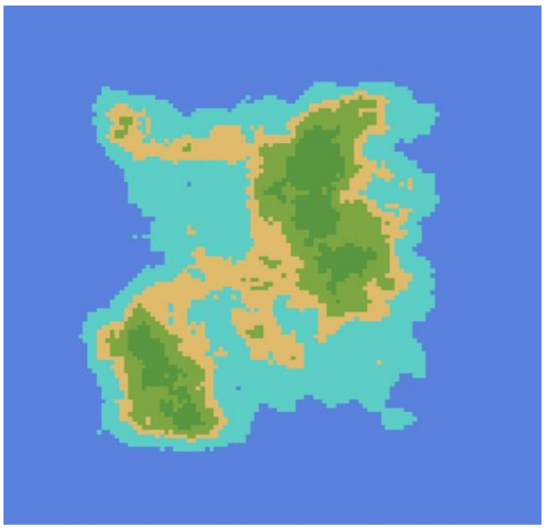
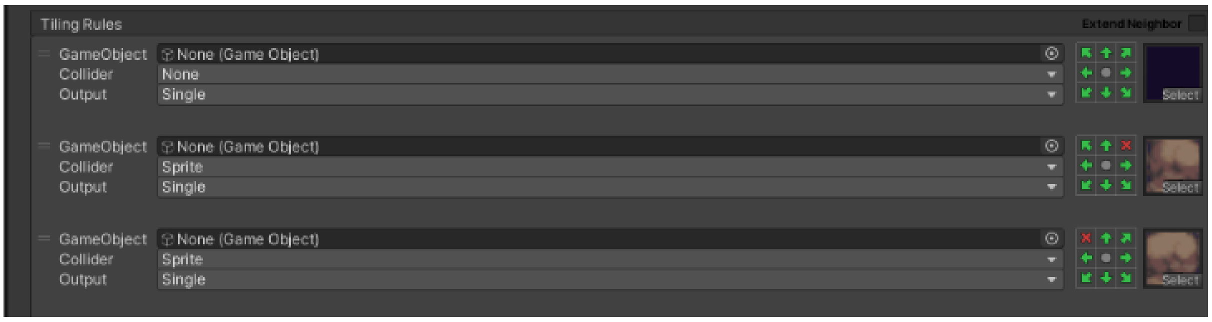


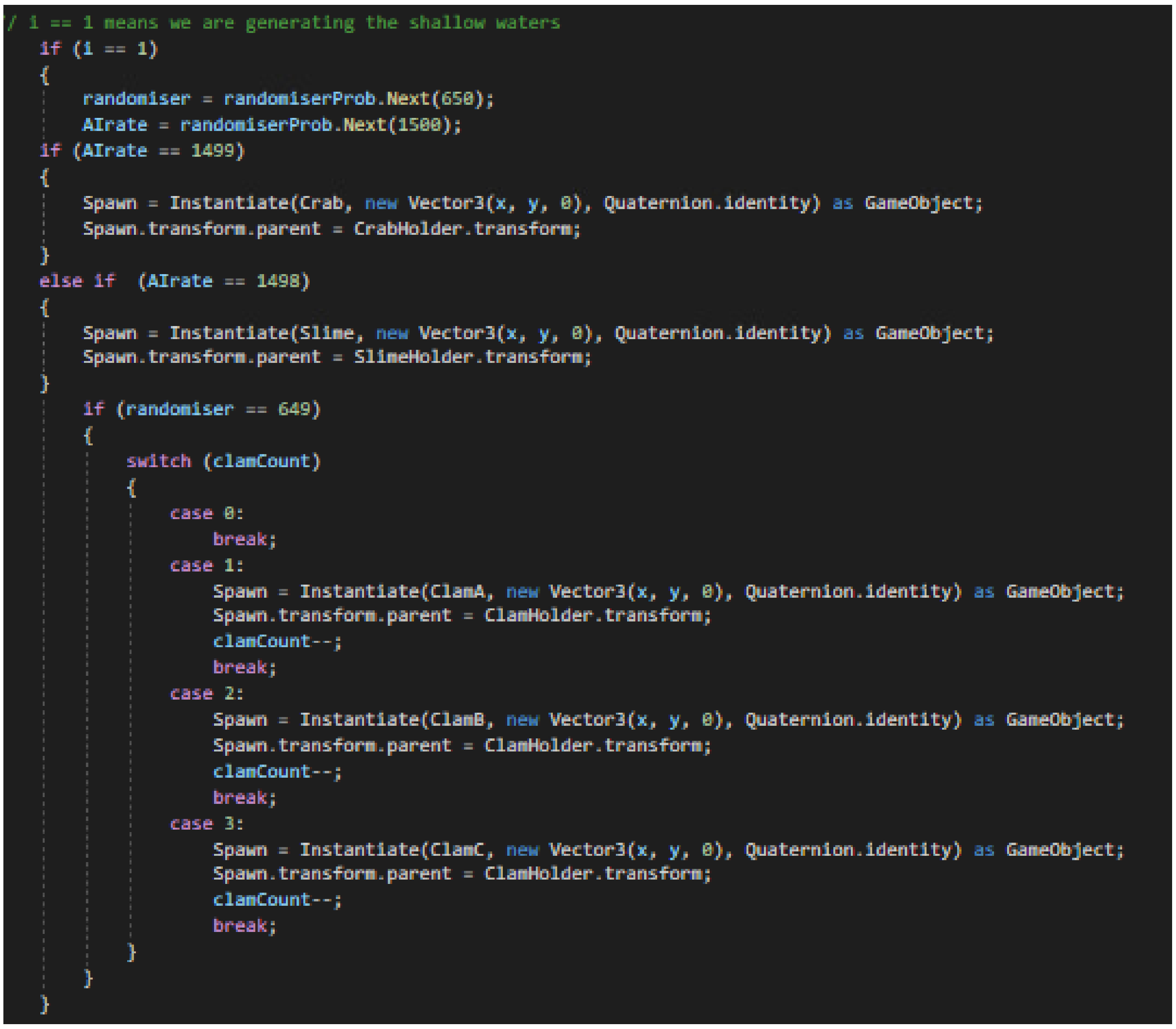
Figure 58 - Perlin Island Example 2



Tiling Rules | Extend Neighbor

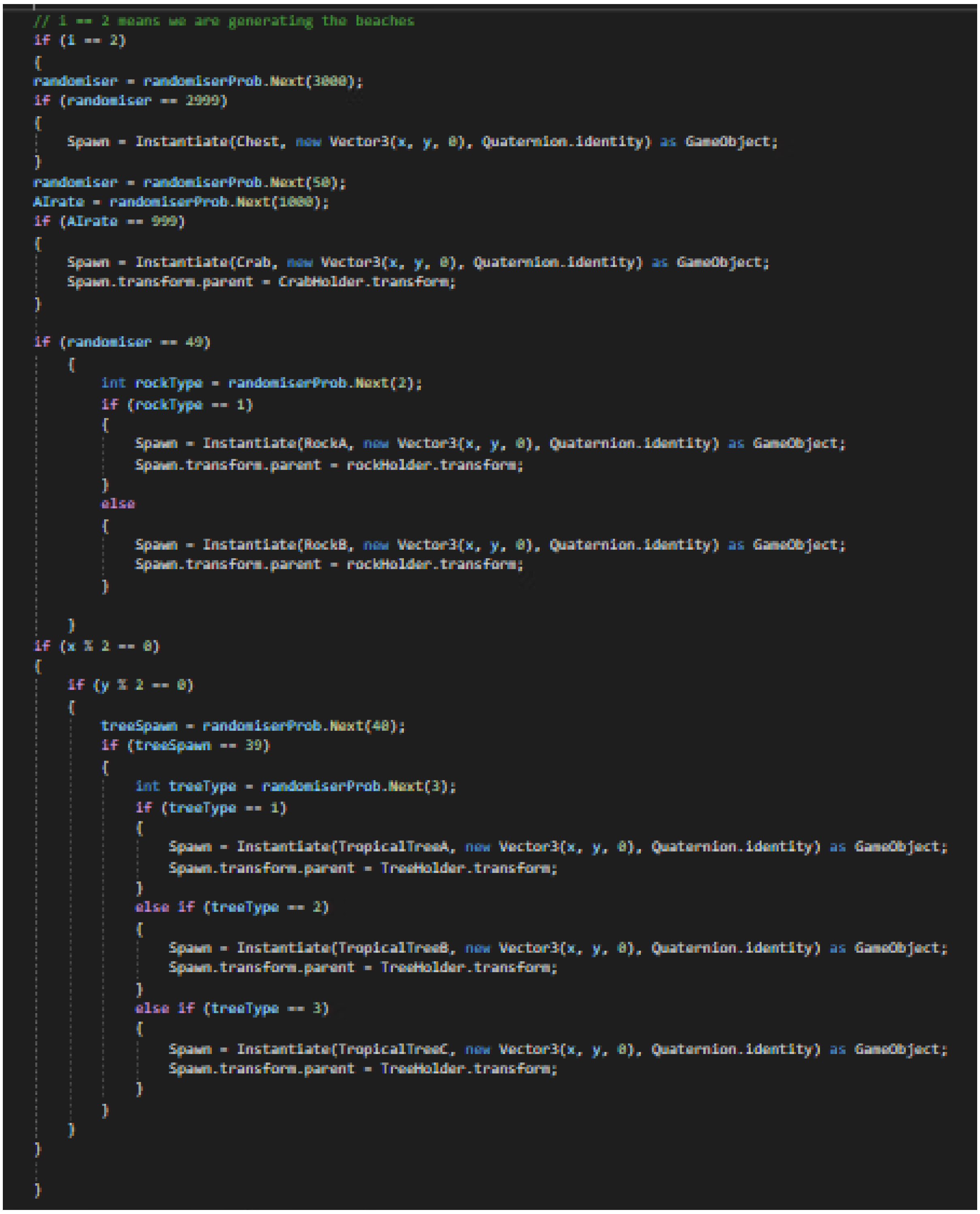
| | |
|---|---|
| GameObject | None (Game Object) |
| Collider | None |
| Output | Single |
| GameObject | None (Game Object) |
| Collider | Sprite |
| Output | Single |
| GameObject | None (Game Object) |
| Collider | Sprite |
| Output | Single |

Figure 60 – Rule Tiles Setup



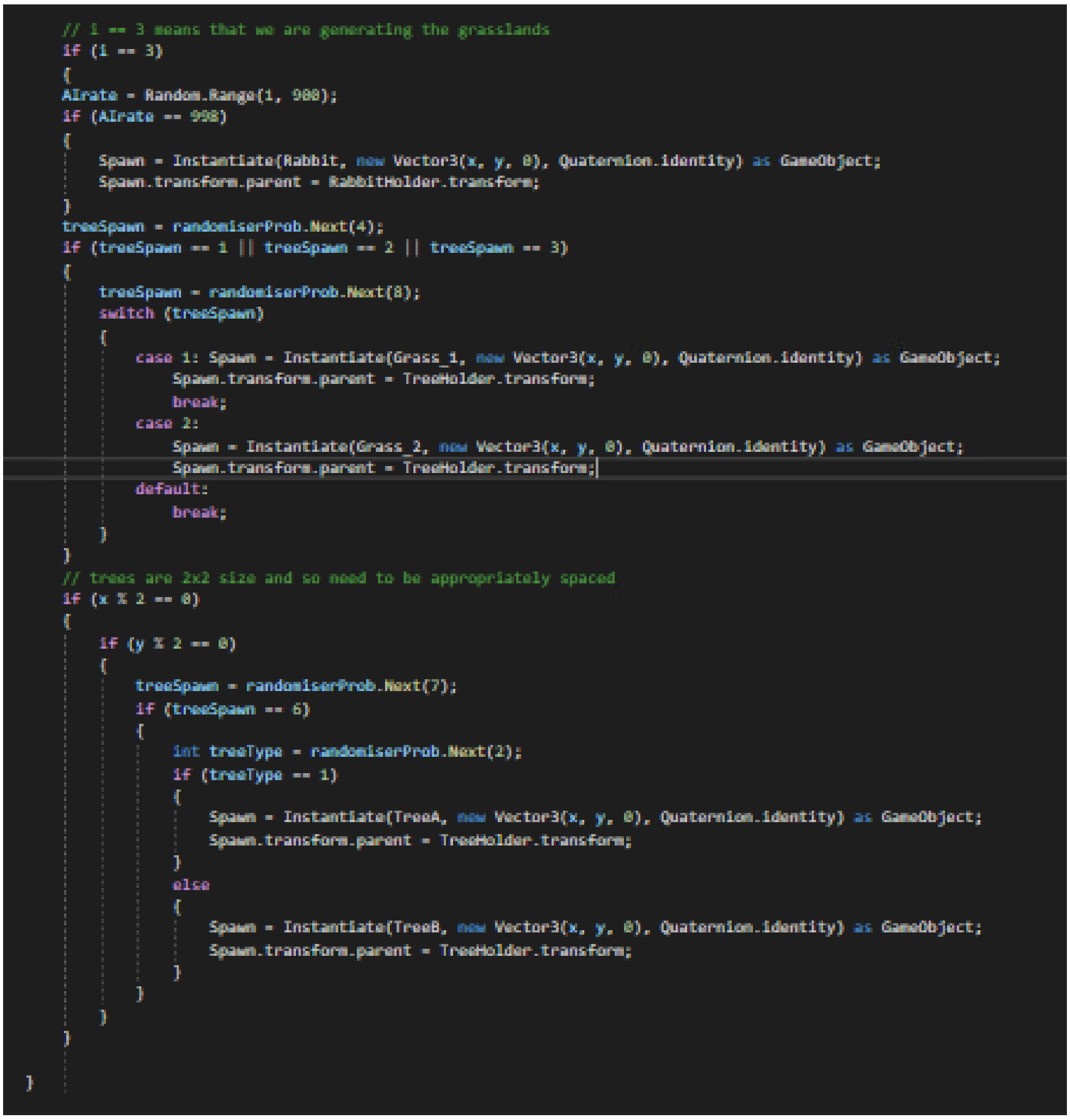
```
/ i == 1 means we are generating the shallow waters
    if (i == 1)
    {
        randomiser = randomiserProb.Next(650);
        AIrate = randomiserProb.Next(1500);
    if (AIrate == 1499)
    {
        Spawn = Instantiate(Crab, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
        Spawn.transform.parent = CrabHolder.transform;
    }
    else if  (AIrate == 1498)
    {
        Spawn = Instantiate(Slime, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
        Spawn.transform.parent = SlimeHolder.transform;
    }
        if (randomiser == 649)
        {
            switch (clamCount)
            {
                case 0:
                    break;
                case 1:
                    Spawn = Instantiate(ClamA, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                    Spawn.transform.parent = ClamHolder.transform;
                    clamCount--;
                    break;
                case 2:
                    Spawn = Instantiate(ClamB, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                    Spawn.transform.parent = ClamHolder.transform;
                    clamCount--;
                    break;
                case 3:
                    Spawn = Instantiate(ClamC, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                    Spawn.transform.parent = ClamHolder.transform;
                    clamCount--;
                    break;
            }
        }
    }
```

Figure 61 - Code Snippet to Generate Shallow Water Biome and Resources



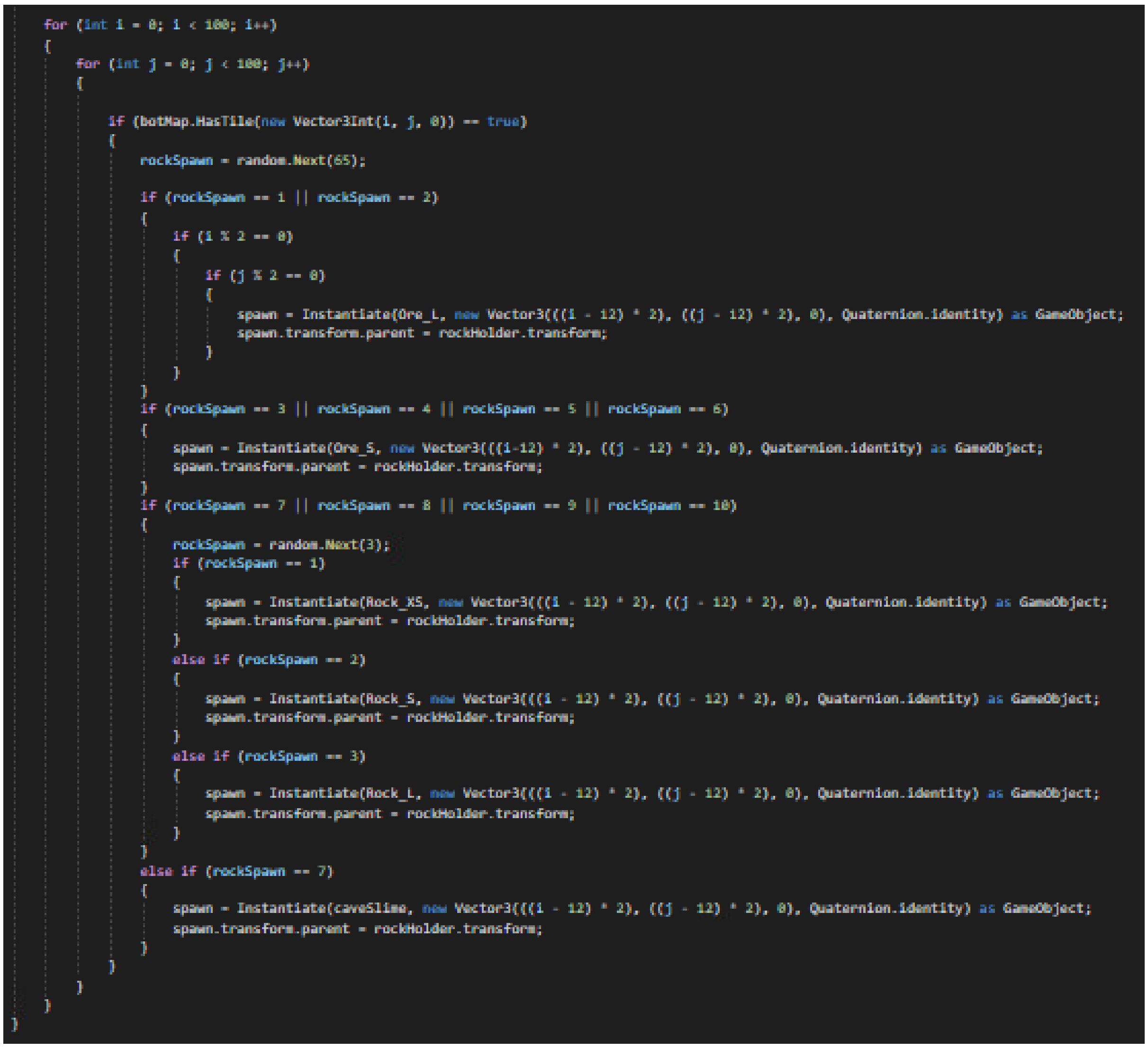
```
// i == 2 means we are generating the beaches
if (i == 2)
{
randomiser = randomiserProb.Next(3000);
if (randomiser == 2999)
{
    Spawn = Instantiate(Chest, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
}
randomiser = randomiserProb.Next(50);
AIrate = randomiserProb.Next(1000);
if (AIrate == 999)
{
    Spawn = Instantiate(Crab, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
    Spawn.transform.parent = CrabHolder.transform;
}

if (randomiser == 49)
    {
        int rockType = randomiserProb.Next(2);
        if (rockType == 1)
        {
            Spawn = Instantiate(RockA, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
            Spawn.transform.parent = rockHolder.transform;
        }
        else
        {
            Spawn = Instantiate(RockB, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
            Spawn.transform.parent = rockHolder.transform;
        }

    }
if (x % 2 == 0)
{
    if (y % 2 == 0)
    {
        treeSpawn = randomiserProb.Next(40);
        if (treeSpawn == 39)
        {
            int treeType = randomiserProb.Next(3);
            if (treeType == 1)
            {
                Spawn = Instantiate(TropicalTreeA, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                Spawn.transform.parent = TreeHolder.transform;
            }
            else if (treeType == 2)
            {
                Spawn = Instantiate(TropicalTreeB, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                Spawn.transform.parent = TreeHolder.transform;
            }
            else if (treeType == 3)
            {
                Spawn = Instantiate(TropicalTreeC, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                Spawn.transform.parent = TreeHolder.transform;
            }
        }
    }
}


}
```

Figure 62 - Code Snippet to Generate Beach Biome and resources



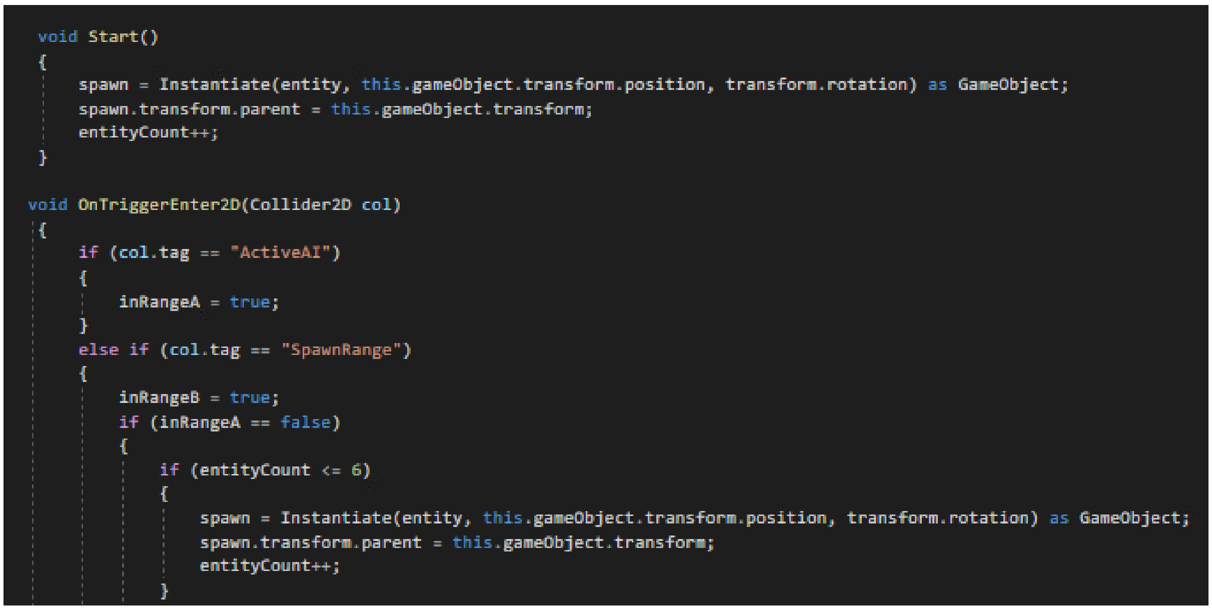
```
    // i == 3 means that we are generating the grasslands
    if (i == 3)
    {
    AIrate = Random.Range(1, 900);
    if (AIrate == 998)
    {
        Spawn = Instantiate(Rabbit, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
        Spawn.transform.parent = RabbitHolder.transform;
    }
    treeSpawn = randomiserProb.Next(4);
    if (treeSpawn == 1 || treeSpawn == 2 || treeSpawn == 3)
    {
        treeSpawn = randomiserProb.Next(8);
        switch (treeSpawn)
        {
            case 1: Spawn = Instantiate(Grass_1, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                Spawn.transform.parent = TreeHolder.transform;
                break;
            case 2:
                Spawn = Instantiate(Grass_2, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                Spawn.transform.parent = TreeHolder.transform;
            default:
                break;
        }
    }
    // trees are 2x2 size and so need to be appropriately spaced
    if (x % 2 == 0)
    {
        if (y % 2 == 0)
        {
            treeSpawn = randomiserProb.Next(7);
            if (treeSpawn == 6)
            {
                int treeType = randomiserProb.Next(2);
                if (treeType == 1)
                {
                    Spawn = Instantiate(TreeA, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                    Spawn.transform.parent = TreeHolder.transform;
                }
                else
                {
                    Spawn = Instantiate(TreeB, new Vector3(x, y, 0), Quaternion.identity) as GameObject;
                    Spawn.transform.parent = TreeHolder.transform;
                }
            }
        }
    }

}
```

Figure 63 - Code Snippet to Generate Grassland Biome and Resources



```
    for (int i = 0; i < 100; i++)
    {
        for (int j = 0; j < 100; j++)
        {

            if (botMap.HasTile(new Vector3Int(i, j, 0)) == true)
            {
                rockSpawn = random.Next(65);

                if (rockSpawn == 1 || rockSpawn == 2)
                {
                    if (i % 2 == 0)
                    {
                        if (j % 2 == 0)
                        {
                            spawn = Instantiate(Ore_L, new Vector3(((i - 12) * 2), ((j - 12) * 2), 0), Quaternion.identity) as GameObject;
                            spawn.transform.parent = rockHolder.transform;
                        }
                    }
                }
                if (rockSpawn == 3 || rockSpawn == 4 || rockSpawn == 5 || rockSpawn == 6)
                {
                    spawn = Instantiate(Ore_S, new Vector3(((i-12) * 2), ((j - 12) * 2), 0), Quaternion.identity) as GameObject;
                    spawn.transform.parent = rockHolder.transform;
                }
                if (rockSpawn == 7 || rockSpawn == 8 || rockSpawn == 9 || rockSpawn == 10)
                {
                    rockSpawn = random.Next(3);
                    if (rockSpawn == 1)
                    {
                        spawn = Instantiate(Rock_XS, new Vector3(((i - 12) * 2), ((j - 12) * 2), 0), Quaternion.identity) as GameObject;
                        spawn.transform.parent = rockHolder.transform;
                    }
                    else if (rockSpawn == 2)
                    {
                        spawn = Instantiate(Rock_S, new Vector3(((i - 12) * 2), ((j - 12) * 2), 0), Quaternion.identity) as GameObject;
                        spawn.transform.parent = rockHolder.transform;
                    }
                    else if (rockSpawn == 3)
                    {
                        spawn = Instantiate(Rock_L, new Vector3(((i - 12) * 2), ((j - 12) * 2), 0), Quaternion.identity) as GameObject;
                        spawn.transform.parent = rockHolder.transform;
                    }
                }
                else if (rockSpawn == 7)
                {
                    spawn = Instantiate(caveSlime, new Vector3(((i - 12) * 2), ((j - 12) * 2), 0), Quaternion.identity) as GameObject;
                    spawn.transform.parent = rockHolder.transform;
                }
            }
        }
    }
}
```

Figure 65 - Code Snippet Showing the Cellular Automata Resource Generation



```
  void Start()
  {
      spawn = Instantiate(entity, this.gameObject.transform.position, transform.rotation) as GameObject;
      spawn.transform.parent = this.gameObject.transform;
      entityCount++;
  }

void OnTriggerEnter2D(Collider2D col)
  {
      if (col.tag == "ActiveAI")
      {
          inRangeA = true;
      }
      else if (col.tag == "SpawnRange")
      {
          inRangeB = true;
          if (inRangeA == false)
          {
              if (entityCount <= 6)
              {
                  spawn = Instantiate(entity, this.gameObject.transform.position, transform.rotation) as GameObject;
                  spawn.transform.parent = this.gameObject.transform;
                  entityCount++;
              }
```

Figure 64 - Code Snippet to Show AI Spawner Functionality

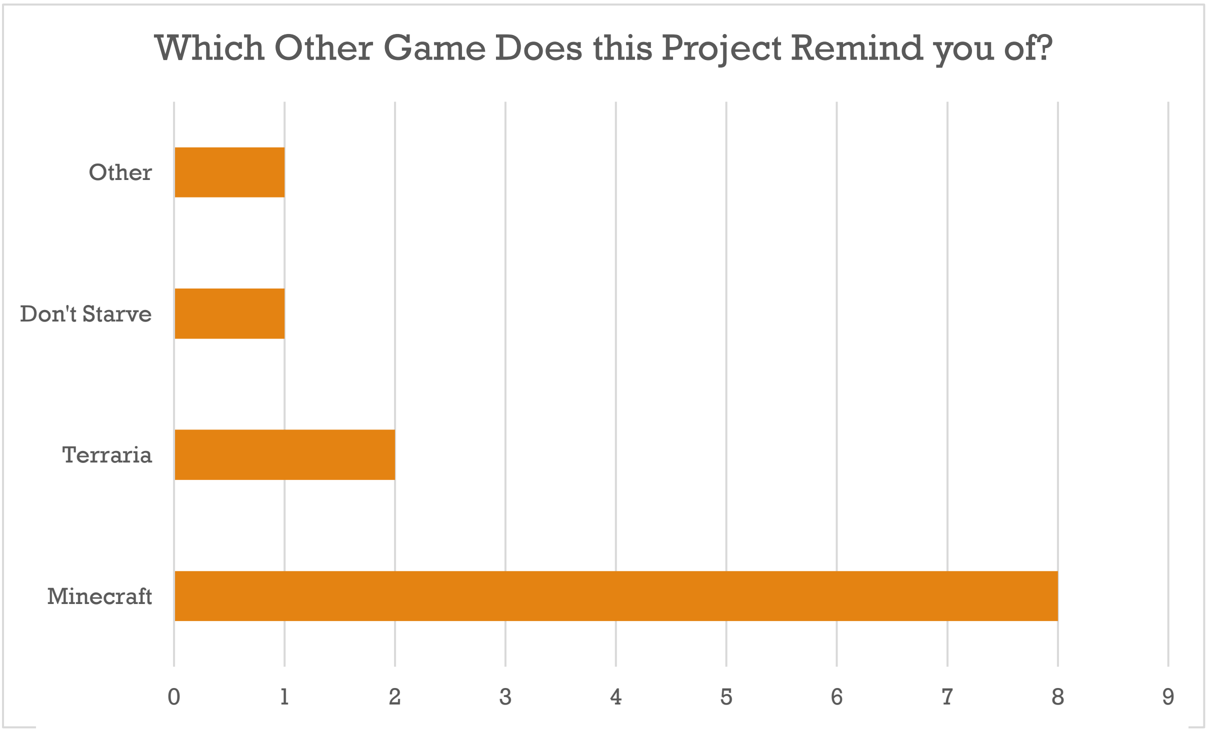


Figure 66 - Table to Show Which Games hosTILE Resembles

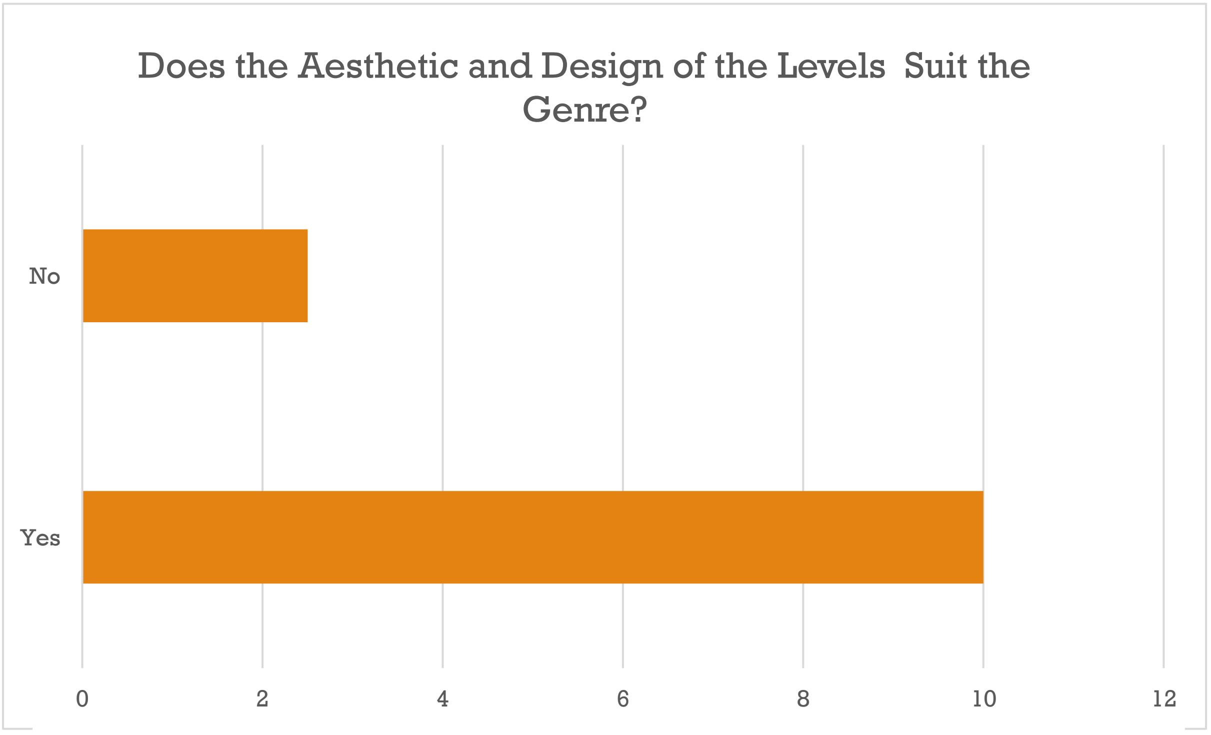


Figure 67 - Table to Show Opinions on the Game's Aesthetic Fitting the Genre's Conventional Image

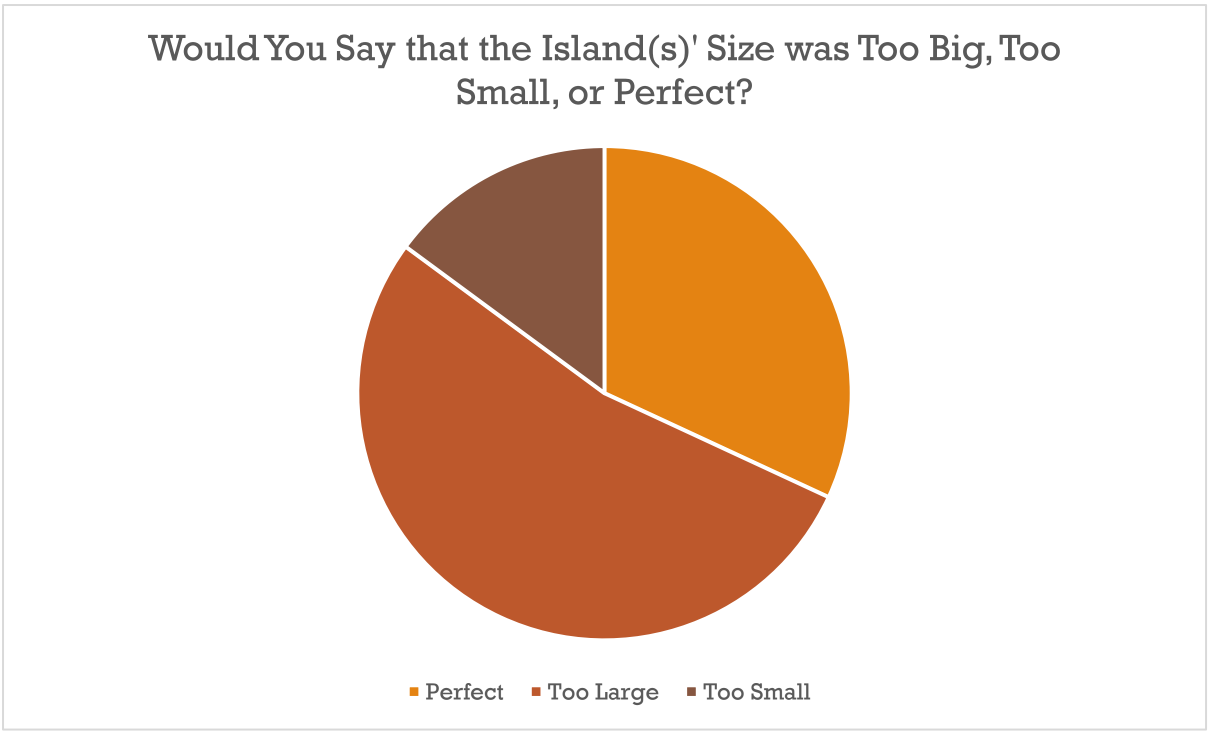


Figure 68 - Table to Show the Opinion on the Size of the Island(s)

## REFERENCES

BIT-101. 2022. *Perlin vs. Simplex*. [online] Available at: <https://www.bit-101.com/blog/2021/07/perlin-vs-simplex/> [Accessed 3 February 2022].

Brill, F., 1997. Perception and Action in a Dynamic Three-Dimensional World, p17.

Caballero, L., Hodge, B. and Hernandez, S., 2016. Conway's "Game of Life" and the Epigenetic Principle. *Frontiers in Cellular and Infection Microbiology*, 6(1), p.1.

Cook, M., Colton, S., Gow, J. and Smith, G., 2019. *General Analytical Techniques For Parameter-Based Procedural Content Generators*. [online] Ieee-cog.org. Available at: <https://ieee-cog.org/2019/papers/paper_84.pdf> [Accessed 22 January 2022].

ebookreading. 2022. *The Pros and Cons of Procedural Generation*. [online] Available at: <https://ebookreading.net/view/book/EB9781785886713_107.html> [Accessed 5 May 2022].

García-Morales, V., 2012. *Symmetrical Von Neumann, hexagonal and Moore neighborhoods in two dimensions*. [online] ResearchGate. Available at: <https://www.researchgate.net/figure/Symmetrical-Von-Neumann-hexagonal-and-Moore-neighborhoods-in-two-dimensions-A_fig6_257191841> [Accessed 16 May 2022].

Gonzalez Vivo, P. and Lowe, J., 2022. *The Book of Shaders*. [online] The Book of Shaders. Available at: <https://thebookofshaders.com/12/> [Accessed 8 February 2022].

Green, S., 2022. *Chapter 26. Implementing Improved Perlin Noise*. [online] NVIDIA Developer. Available at: <https://developer.nvidia.com/gpugems/gpugems2/part-iii-high-quality-rendering/chapter-26-implementing-improved-perlin-noise> [Accessed 21 January 2022].

Himite, B., 2021. *Replicating Minecraft World Generation in Python*. [online] Medium. Available at: <https://towardsdatascience.com/replicating-minecraft-world-generation-in-python-1b491bc9b9a4> [Accessed 22 January 2022].

Jaokar, A., 2019. *3 Main Approaches to Machine Learning Models - KDnuggets*. [online] KDnuggets. Available at: <https://www.kdnuggets.com/2019/06/main-approaches-machine-learning-models.html> [Accessed 26 March 2022].

Kowalski, K., 2020. *Game Development Tutorial | Cellular Automata and Procedural Map Generation*. [online] Youtube.com. Available at: <https://www.youtube.com/watch?v=slTEz6555Ts> [Accessed 25 March 2022].

Linguazza.com. 2014. *SURVIVAL GAME definition*. [online] Available at: <https://linguazza.com/definition/survival+game> [Accessed 5 May 2022].

Millington, I. and Funge, J., 2009. *Artificial intelligence for games*. Elsevier inc., p.9.

Muñoz-Avila, H., n.d. *Meaningful Play and Game Design*. [online] studylib.net. Available at: <https://studylib.net/doc/9249708/meaningful-play-and-game-design> [Accessed 27 March 2022].

Parberry, I., 2014. *Designer Worlds: Procedural Generation of Infinite Terrain from Real-World Elevation Data*. [online] jcgt.org. Available at: <https://jcgt.org/published/0003/01/04/paper.pdf> [Accessed 21 January 2022].

Pedersen, K., 2014. *Procedural Level Generation in Games using a Cellular Automaton: Part 1*. [online] raywenderlich.com. Available at: <https://www.raywenderlich.com/2425-procedural-level-generation-in-games-using-a-cellular-automaton-part-1> [Accessed 22 January 2022].

Plarium.com. 2022. *Survival Games: A Guide to the Classic Gaming Genre*. [online] Available at: <https://plarium.com/en/blog/survival-games/> [Accessed 12 February 2022].

Plato.stanford.edu. 2017. *Cellular Automata (Stanford Encyclopedia of Philosophy)*. [online] Available at: <https://plato.stanford.edu/entries/cellular-automata/> [Accessed 22 January 2022].

Robins, A., 2020. *Stochastic vs Deterministic Models: Understand the Pros and Cons*. [online] Blog.ev.uk. Available at: <https://blog.ev.uk/stochastic-vs-deterministic-models-understand-the-pros-and-cons> [Accessed 27 March 2022].

Scher, Y., 2017. *Playing with Perlin Noise: Generating Realistic Archipelagos*. [online] Medium. Available at: <https://medium.com/@yvanscher/playing-with-perlin-noise-generating-realistic-archipelagos-b59f004d8401> [Accessed 22 January 2022].

Schmidt, A., 2022. *Random Walks*. [online] Mit.edu. Available at: <https://www.mit.edu/~kardar/teaching/projects/chemotaxis(AndreaSchmidt)/random.htm> [Accessed 12 February 2022].

Sylvester, T., 2013. *Designing Games*. [online] O’Reilly Online Learning. Available at: <https://www.oreilly.com/library/view/designing-games/9781449338015/ch01.html> [Accessed 19 May 2022].

Techopedia.com. 2019. *What is a Non-Deterministic Algorithm? - Definition from Techopedia*. [online] Available at: <https://www.techopedia.com/definition/24618/non-deterministic-algorithm> [Accessed 27 March 2022].

unwttng.com. 2018. *Procedural Generation - How Does It Work? | The website of Juniper Preston, Computerer*. [online] Available at: <https://unwttng.com/how-does-procedural-generation-work-random-noise> [Accessed 24 March 2022].

Wolfram, S., 2002. *Historical Notes from Stephen Wolfram's A New Kind of Science*. [online] Wolframscience.com. Available at: <https://www.wolframscience.com/reference/notes/876b> [Accessed 22 January 2022].

World-machine.com. 2022. *Device Reference*. [online] Available at: <http://www.world-machine.com/learn.php?page=devref> [Accessed 18 March 2022].

# APPENDIX

## THIRD PARTY ASSETS

### GRAPHICAL ASSETS

#### TILE SET

https://pita.itch.io/rpg-dungeon-tileset

https://pita.itch.io/rpg-overworld-tileset

https://pita.itch.io/rpg-village-tileset

#### ITEM SPRITES AND ANIMALS

https://szadiart.itch.io/craftland

https://elthen.itch.io/2d-pixel-art-crab-sprites

### SOUND EFFECTS

https://freesound.org/people/DrMaysta/sounds/418509/

https://freesound.org/people/Q.K./sounds/56271/

https://freesound.org/people/SilverIllusionist/sounds/411178/

https://freesound.org/people/Raclure/sounds/483598/

https://freesound.org/people/JanKoehl/sounds/85581/

https://freesound.org/people/ultraaxvii/sounds/591152/

https://freesound.org/people/igroglaz/sounds/593875/

https://freesound.org/people/zimbot/sounds/244490/

https://freesound.org/people/XxChr0nosxX/sounds/268227/

https://freesound.org/people/deleted_user_2104797/sounds/164678/

https://freesound.org/people/jorickhoofd/sounds/161593/

https://freesound.org/people/super8ude/sounds/442538/

https://freesound.org/people/Debsound/sounds/168822/

https://freesound.org/people/zimbot/sounds/122126/

https://freesound.org/people/Flying_Deer_Fx/sounds/369010/

https://freesound.org/people/MadPanCake/sounds/567849/