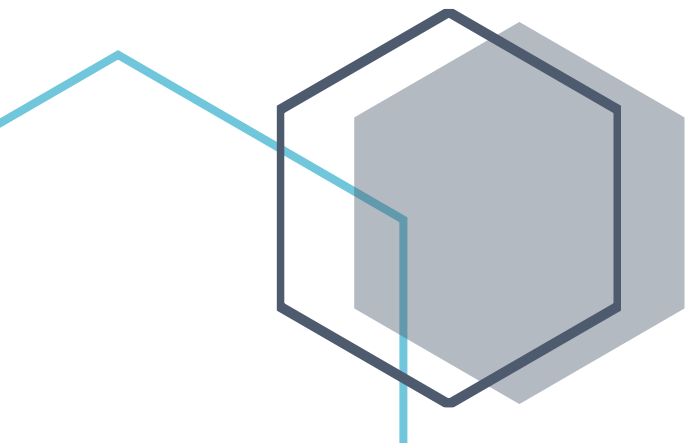
Procedural Generation
NEA Project 2018/19

TOM ALNER - 7219
CANFORD SCHOOL – 55243



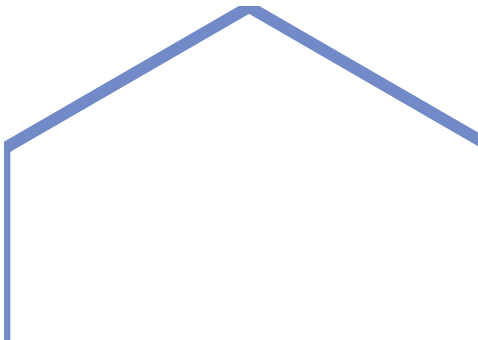


Table of Contents

Analysis 5

Introduction 5

Project Statement: 5

Identification of User Needs: 6

Research: 7

Terrain Generation 7

Meshes: 8

-Lighting and Flatshading: 9

Enemies 10

Third Person Controller 10

Existing Systems 11

Portal Knights: Procedurally Generated Characteristic Terrain, Player HUD, Inventory, 3rd Person Combat and Camera System, Island Returnability 11

Zelda Breath of the Wild: Detection System, Player HUD, Inventory, 3rd Person Combat and Camera System 12

Prototyping: 13

Camera System: 13

Poisson Disc Sampling: 16

Data Volume: 22

Objectives: 22

Design 25

Overview 25

Visualisation 25

Hierarchy: 25

Class Definitions (UML Diagram) 26
Data 27
Data Dictionary: Player Class 28
Data Dictionary: Noise Generator Class 28
Algorithms 29
Noise-Map Generation: 29
Mesh Generation: 31
Falloff-Map Generation: 32
Object Placement: 35
Pathfinding: 39
Abstract Data 45
2D Array 45
List 45
Dictionary 45
Queue 45
Coordinate 45
TerrainType 46
PoissonObject 46
Unity 46
UI Design 48
Game Scene: 49
Inventory Menu: 49
Main Menu: 51
Settings Menu: 51
Solution Development 53

Annotated Code Listing 53

Map Generation 53

Item Functionality 78

Camera 79

Player and Enemies 82

Pathfinding 89

UI 100

General 103

Testing and Evaluation 109

Test Plan 109

Erroneous Data 120

Testing Evidence 121

Objective 1 121

Objective 2 127

Objective 3 127

Objective 4 129

Objective 5 130

Objective 6 131

Objective 8 132

Evaluation 133

Evaluation Against Objectives 133

Objective 1: **Procedurally generate random, traversable and characteristic terrain using a 3D noise function with applied variables.** 133

Objective 2: **Randomly spawn natural biomes onto the terrain in suitable and realistic locations.** 134

Objective 3: **Create a detection system for the enemies such that they can locate and approach the player.** 135

Objective 4: **IMPLEMENT AN INTERACTIVE 3RD PERSON CAMERA SYSTEM.** 136

Objective 5: **SPAWN A RANGE OF COLLECTABLE ITEMS IN THE WORLD TO ASSIST THE PLAYER IN TRAVERSAL OR COMBAT.** 136

Objective 6: **CREATE A SIMPLISTIC AND INFORMATIVE UI TO DISPLAY GAME INFORMATION.** 138

Extension Objectives 138

Effectiveness of Solution 139

End User Feedback 141

Summary of End-User Feedback 141

Completed Improvements 142

Settings Menu: 143

Potential Improvements 143

Bibliography 145

Terrain Generation 145

Object Placement 145

Pathfinding 145

Analysis

INTRODUCTION

Most modern games manually create large-scale maps upon which the game is based. These games are most commonly produced by large companies or development teams who allocate people, time and resources into developing a suitable and impressive environment for their game, as the environment is considered a crucial part in many games and can be instrumental in its success or failure. However, as a lone person it isn't feasible for me to manually create a large map as a base for a game, it would take a huge amount of time and would require software with no relevance to programming. By creating an algorithm to procedurally generate terrain it would eliminate the need for me to manually place objects in the world. Furthermore, with development I would be able to create endless terrain, something which couldn't be manually achieved.

Once an algorithm has been created, by adjusting the variables used to create the terrain, I could instantly manipulate the land to shape as I want, given that the algorithm has the required complexity and user influence. With this it would be possible to create different styles of land by manipulating the terrain. I will focus on this concept as I feel that it has lots of scope for improvement and complexity, with a variety of interesting algorithmic approaches that I can explore. These variables can then be used to determine the spawn of natural features that relate to the type of land being created.

I think that an exploration styled game would allow me to make the most of such an algorithm, as the player can experience all the different types of terrain. I plan to implement the game using a 3D game engine, this will assist me in the 3D graphical aspect, and will allow me to place pre-created models in the game, rather than having to create my own.

The game would most likely be suitable for a PC or console, as the creation of the meshes with thousands of vertices and multiple textures over a large area requires lots of processing power. I feel as though a graphics card may be required on devices on which the game runs so a mobile port wouldn't be suitable, as the game may be too strenuous on most mobile devices. Furthermore, a separate UI would need to be created, and the controls would likely be too plentiful/ complex to transfer to a mobile device. Such, I feel that a large open-world map in my project would be best enjoyed by a player on PC where they can easily interact with and observe their surroundings, on which the game is built upon.

PROJECT STATEMENT:

I will create my project such that it can make the most out of the procedurally generated terrain, as that is the area which I believe has the most scope for development and complexity. I plan to spawn the player on an island with many different distinctive types of terrain e.g. grasslands, desert, forests, mountains. I will create these different types of terrain by modifying the variables used in the creation of the terrain such that it reflects the real-life characteristics of that type of area. I will create a goal collectable somewhere on the terrain that the player must find in order to win. The goal item may be difficult to find -due to the large scale of the map - and so I will spawn a range of helper collectables across the map to make this task easier. These items will range in how significant their abilities are,

giving the player a sense of excitement and anticipation before they pick one up. Additionally, such a game forces the player to explore all the different types of environment.

The map, however, will be populated with enemies that can defeat the player, forcing them to restart. These enemies will have a realistic detection algorithm, such that they can only locate the player when they are within a certain distance and within the enemy's peripherals, furthermore the player cannot be detected if they sufficiently obscured by an object. Once an enemy has detected the player, the enemy should move towards the player.

The player should have a menu where they can view an image showing the whole map. In this menu the player should be able to see the effects of the objects that they have collected so far. Another UI element that I hope to achieve is a player HUD, this should display the player's stats such as health and also a live minimap. This should help me to improve the aesthetic and functionality of the game.

The player will be able to revisit previous islands from earlier in the game, this may be to gather resources, or other items that they require which they remember finding at the island. This will require all aspects of each island to be saved in an accessible location. This may require the use of a database due to the volume of data required. Although each player would have to have a personal database, or else a server would be required to store the information of every single player. This will require interaction between the game and the database using SQL or PHP.

IDENTIFICATION OF USER NEEDS:

My prospective user is Ben.E, a student in my year at school. He has played games that have aspects which I hope to work on in my project. I spoke to him about my initial ideas for the project, and he helped me identify they key broad objectives which I should focus and base my project on. He said that the UI should be more of a focus than the combat system as it requires more programming functionality than combat, which requires focus in modelling and animation. Therefore, he felt that improving a basic combat system would be better as an extension objective, as it would make more sense for the HUD to be an integral part, as the game wouldn't be very functional if the player couldn't easily view their stats. He also thought that allowing players to revisit previous islands wouldn't need to be an integral part of the game, and that time spent implementing it could reap more noticeable rewards in other areas of the project.

1. Procedurally generate random, traversable and characteristic terrain using a 2D noise function with applied variables.
2. Randomly spawn natural biomes onto the terrain in suitable and realistic locations.
3. Create a detection system for the enemies such that they can locate and approach the player
4. Implement an interactive 3rd person camera system.
5. Spawn a range of collectable game items to assist the player in traversal or combat.
6. Create a simplistic and informative player HUD to display user stats.
7. [Extension] Create an interactive and skill-based combat system.
8. [Extension] Allow players to revisit previous islands.

RESEARCH:

From my research I hope to discover - or expand my knowledge on - techniques and algorithms that I can use to achieve the concepts in my project statement. I am focusing most of my research onto the terrain generation, this is because it is the basis of my project and is something which I have no experience in.

Terrain Generation

- Noise:

Currently a standard an effective way of generating 'random' terrain is by using noise. The most commonly used and first of these is Perlin Noise. In terms of terrain generation, I can take a 2D slice of Perlin Noise at z = 0 for example to give me a 2D grid of pseudorandom values. I can then map the grid onto a plane and use the value at each (x,y) coordinate as a height giving me a 3D terrain.

- Octaves, Lacunarity and Persistence:

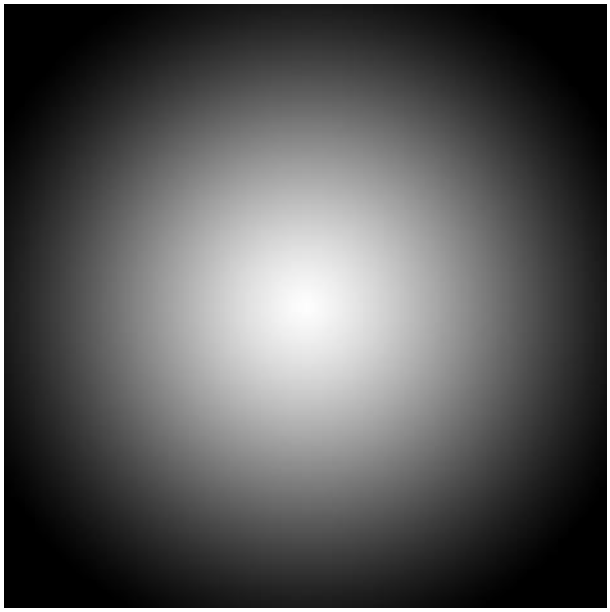
Octaves allow for the increase in detail of the procedurally generated terrain. Each octave is a level of noise, by overlaying these levels of noise on top of each other, it makes the terrain less smooth and more natural looking. Each octave must differ from the previous, else the noise would just increase in amplitude.

Two variables known as lacunarity and persistence are used to influence the noise between octaves.

Lacunarity is used to influence the frequency of the noise, it is a multiplier that determines how quickly frequency increases with each successive octave. When overlaying octaves to create terrain, the frequency of the noise used in each octave should increase, thus with each octave the level and amount of detail will increase. The frequency in each octave could be defined as lacunarity^(n-1) where n is the index of the octave. Lacunarity should therefore be greater than 1 such that the frequency of the noise increases with each octave.

Persistence is used to influence the amplitude of the noise, it is a multiplier which determines how quickly the amplitude diminishes with each successive octave. As the frequency of each octave increases, its influence on the terrain should decrease, and as this variable will control the amplitude of each octave, it should decrease with each octave. We can model amplitude in each octave as persistence ^ (n-1), with a persistence value between 0 and 1, the amplitude of the noise will decrease with each successive octave.

- Falloff Map:



Pictured is an example of a falloff map. The falloff map could be applied over the noise generated on a plane, where the value illustrated by the falloff map at a point is subtracted from the noise height value at the same point on the plane. White could represent 0 and black 1, then with a standardised solid noise produced by the C# noise module, the result of the calculation would be between 1 and -1. This could then be standardised using the InverseLerp function or other linear interpolation. The purpose of the falloff-map in a procedurally generated world is to guarantee that an island will be generated, as the edge of the map is completely black, meaning that the resultant height when applied to standardised Perlin Noise, will always be 0.

- Seed and Offset:

The seed is a number used to generate a large random number that will act as an offset to the noise. This offset is applied in both the x and y axis of the noise. This basically allows the user to take a different slice of the noise, without changing its persistence and lacunarity. However, the importance of the seed is that the number generated from it is the same, so if a user changes the seed, but then returns to the previous seed, the noise generated will be the same as before.

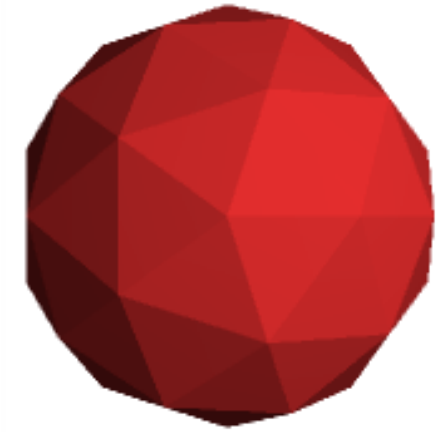
In addition, by giving myself adjustable offset variables in both the x and y directions, I can manually select an area of Perlin Noise.

Meshes:

A mesh is a polygonal approximation of a 3D object. With a plane of noise values, we have values for all 3 axes, if we use the noise value as a height. A mesh is created by making 3D vertices at known locations of the object, these locations would be found using the coordinates on the plane and their corresponding height value. The vertices are then connected to those around them to form a triangle. These triangles together in 3D space create a polygonal object.

-Lighting and Flatshading:

The lighting of a mesh is calculated based on the normals of the vertices in the mesh. The lighting of each triangle is determined by its own vertices' normals; thus, a triangle would only be lighted equally all over if all 3 of its vertices' normals pointed in the same direction. However, in a mesh, most vertices are shared by multiple triangles, thus the normal of that vertex will point in a direction that is a blend of the directions that the triangles that it makes up face. This gives meshes a realistic lighting look.

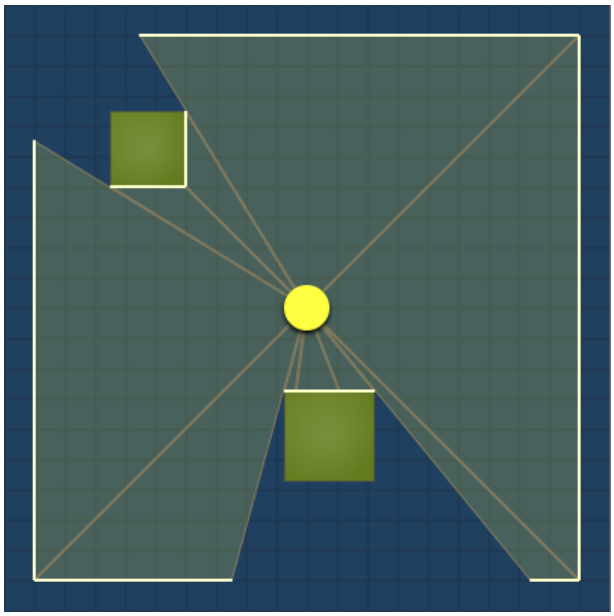


Flatshading is a rendering style where despite a vertex being shared by multiple triangles, is considered as separate in each triangles' lighting such that its normal is simply the direction that the triangle being lit faces. This means that the normals of all the vertices in each triangle will point in the same direction, thus the lighting on each triangle is equal across it. This technique is used in low-poly games, an art style where objects are simplified and polygonal.

- Biomes:

Once a mesh has been generated, to make a terrain look realistic, it needs natural features. Natural features that define a type of land such as trees or water are known as biomes. Unlike the terrain itself, nature's tendencies can't be so well reflected by generating random numbers. Water, however, could be applied to a mesh in a band resembling a sea level, such that any part of the mesh below a certain height, will be water.

For the placement of vegetation, a moisture variable could be used. This could be applied by using another noise function on the plane. Then trees could spawn closer and in greater amounts where there is a high concentration of moisture. Alternatively, the height of the terrain could be sued to define the type of object that should spawn in a location. For example, snow being common in higher areas, and sand more common in areas closer to sea level.



Enemies

-Detection and Visibility Algorithm:

Raycasting is a technique where an invisible ray is fired from a point to check if any colliders are in the path of the ray, such as a wall or a character. Unity has a RaycastHit variable to store information about any colliders that the ray may intersect. A raycast can also be given a distance for which to travel from the point at which it is cast. This can be used to give an enemy a view distance, past which they cannot detect the player, and within which, they can only detect the player if the raycast isn't obscured by another object.

The algorithm is then developed such that rays are only cast at the angles where the edges of walls or obstacles begin. This then creates a grid of triangles, where anything within a triangle can be detected by the viewer. With relation to my project, the algorithm could easily be adjusted to give a narrower field of view in the direction that an enemy is facing.

-Combat:

I feel that the combat system in my project should be relatively simplistic, this will give me more time to develop the fundamentals of the game. In Unity, developing an impressive looking 3rd person combat system requires focus in animations and modelling software. This won't demonstrate much programming complexity, but it could be developed as an extension to improve the aesthetic of the game.

Third Person Controller

-Camera Collision:

Given the simplistic aesthetic of my project, I think that using a 3rd person viewpoint would look best. This will require me to create a camera system, which allows the player to easily move the camera around to view the environment. A collision system will be needed for the camera so that it can manoeuvre around obstacles in the world, so that it can keep a focus on the player on another object, without being obscured. This could be achieved by the aforementioned raycasting technique, the camera's raycast should be able to strike the player, else it should move until it does.

-Camera Focus:

In many 3rd person games, the player has the ability to shift the focus of the camera from themselves to a nearby object or enemy. This is usually indicated by black focus bars. This can be done by changing the camera's 'main focus object' in a 3D game engine, however I will need to calculate the camera's position such that the player themselves is still visible.

EXISTING SYSTEMS

Having developed some initial broad objectives and considered algorithms and concepts that I can use to approach them, I tried to find some existing software which exhibited aspects that I am focusing on in my objectives. I found that there are successful games that procedurally generate terrain, and in some cases animals and plants, but no major titles simply used 2D Perlin Noise to generate height values, all games introduced different algorithms or factors, to achieve their unique objectives and ideals for the game. Just as my project may develop to combine multiple algorithms with the 2D Perlin Noise. The following are games with similar or interesting approaches to some of the concepts that I intend to implement in my project.

Portal Knights: Procedurally Generated Characteristic Terrain, Player HUD, Inventory, 3rd Person Combat and Camera System, Island Returnability

Portal Knights was the newest successful game that I found which exhibited procedurally generated terrain in a way like my ideas. The game has very similar functional aspects to what I hope to achieve, but the style of the game is an online RPG which is different to the survival style that I am aiming for.

I was unable to find the method used to generate the terrain, but the terrain consists of overhangs created by overlaying blocks in 3D space. This seemed very similar to the appearance of the terrain created in Minecraft. I found that this terrain is created by using 3D Perlin Noise, therefore the x,y,z values are already given when finding a Perlin value. In Minecraft The Perlin value found is treated as density, anything greater than 0 is treated as landmass, anything less than 0 is created as air. This allows the sky-islands that people associate with Minecraft and - as I assume - in Portal Knights. As I aim to create sea-level islands surrounded by water, I don't need 3D Perlin Noise as there won't be overhangs, I can simply use the Perlin value as the third-dimension position.

My main reason for researching this game is that it generates characteristic terrain. The player progress through different styles of island. They begin with a grassland, then a desert, then a wooded area. Each island itself is randomly-generated, but each will have the landmarks and objects that reflect the island 'biome'. This is done by each island having a set of assets used to create the terrain. Thus, each player should have a different first island, all the important landmarks will be there, but in different positions. Finding this reassured me that it would be possible to make characteristic terrain as well as being able to return to a perfect recreation of it, however the challenge for me would be blending the types of terrain together. Furthermore, I didn't appear to me that the different styles of terrain had different shapes of land, simply that the characteristic of the land decided the assets and ground blocks that were used to create the island.

As it is an RPG game, a player HUD, inventory system, and 3rd person combat and camera system were present. The HUD was simple, it displayed the player's health and stamina, along with a personal selection of inventory items to allow quick access. The inventory itself is divided into sections, however it feels cluttered and hard to navigate due to the abundance of different collectable items in the world. Regarding the camera, the player can rotate it around themselves, and I am unaware if it can manoeuvre past objects. It does however have a targeting system, so that the player can shift the focus of the camera to an enemy, along with a hotkey to switch to a 1st person camera. The combat system was simple, yet likely more advanced than I will achieve - due to the RPG nature of the game – as different classes are given their typical combat capabilities and weaknesses.

Zelda Breath of the Wild: Detection System, Player HUD, Inventory, 3rd Person Combat and Camera System

Zelda Breath of the Wild is produced by Nintendo, one of the largest electronics and gaming companies in the world, so it is unrealistic for me to achieve a game such as this, but the way in which some of the core gameplay functionality is approached can give me ideas into how to implement my objectives, and the kind of changes which are successful. Despite most Zelda games being in essence a dungeon crawler, BOTW is centred on a large-scale map, however it is manually created. Such, I will look at how core gameplay aspects should operate on a large open world map, by looking at one of the most successful games in the past few years.

It appears that there was detection system in Portal Knights, but being a smaller game, it is hard to tell how it works. In BOTW the enemies are oblivious to a player out of their view distance, or outside of their FOV. The player seems to have a stealth variable, they can wear clothes to increase this or crouch while moving. Furthermore, any sound made by the player will affect how easy it is for an enemy to detect them. If the player is making themselves obvious by attacking or being loud and nearby then the enemy may spot them instantly. Else, a 'detection gauge' appears above the enemy's head, it shows

how close the enemy is to detecting and consequently engaging the player. If the player can remove themselves from sight, then the detection gauge may fall, remaining in the same visible position once the enemy tries to detect will mean that the gauge will slowly fill. While an enemy is trying to detect a player, their behaviour will change, such that they will try to move to where they believe the player might be. The enemies don't appear to move round objects in order to get a better view of the player, they simply move towards where they think the player is, and then move around objects when their path to that location is obstructed. I probably have ignored some of the complexities used in this algorithm, but I believe that I could follow the ideas from the simplified description here to make an effective enemy detection and approach system.

Being a survival styled game, Breath of the Wild has a more expansive HUD than Portal Knights, displaying the player's health, along with a mini-map, temperature, tracker and forecast. I won't need all of these elements in my game, but this was a good demonstration of how a HUD can be displayed concisely. The survival resources are simple in concept, different types of plant give different rewards, and are found in different areas, certainly a concept that I could introduce in my project.

The inventory system is much better than Portal Knights, the player can preview the effect that an item will have on them. Furthermore, the game pauses when the player is accessing the inventory, unlike in Portal Knights, I haven't decided which option I would implement yet. Additionally, I feel that the cel-shaded graphics rendering style is effective in the environment, so I will consider or experiment with its use in my project.

Like most modern Zelda games, there is a very impressive 3rd person camera. The player can orbit the camera around themselves, and it will manoeuvre to keep a view of the player. The player can make the camera target a nearby enemy or NPC, in combat this changes the control system slightly to allow the player to make dodges. The combat system is again more advanced than I will be able to achieve, this is due to the multiple styles of attacks that result from fighting different enemies, and the relationship that these attacks have with the timing of the player's actions. I will be limited in this field due to not being able to create hundreds of animations and then determine the result when they are combined in the game space.

PROTOTYPING:

Camera System:

Camera Follow Script:

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

public class CameraFollow : MonoBehaviour
{

        public float CameraMoveSpeed = 120.0f;
        public GameObject CameraFollowObject;
        Vector3 FollowPos;
        public float clampAngle = 80.0f;
        public float inputSensitivity = 150.0f;
        public GameObject PlayerObj;
```

```
        public float mouseX;
        public float mouseY;
        public float finalInputX;
        public float finalInputZ;
        public float rotY = 0.0f;
        public float rotX = 0.0f;
        public float smooth = 10.0f;


        // Use this for initialization
        void Start()
        {
                Vector3 rot = transform.localRotation.eulerAngles;
                rotY = rot.y;
                rotX = rot.x;
                Cursor.lockState = CursorLockMode.Locked;
                Cursor.visible = false;
        }

        // Update is called once per frame
        void Update()
        {
                float inputX = Input.GetAxis("RightStickHorizontal");
                float inputZ = Input.GetAxis("RightStickVertical");
                mouseX = Input.GetAxis("Mouse X");
                mouseY = Input.GetAxis("Mouse Y");
                finalInputX = mouseX + inputX;
                finalInputZ = mouseY + inputZ;

                rotY += finalInputX * inputSensitivity * Time.deltaTime;
                rotX += finalInputZ * inputSensitivity * Time.deltaTime;

                rotX = Mathf.Clamp(rotX, -clampAngle, clampAngle);
                Quaternion localRotation = Quaternion.Euler(rotX, rotY, 0.0f);
                transform.rotation = localRotation;

        }

        void LateUpdate()
        {
                CameraStepUpdate();
        }

        void CameraStepUpdate()
        {
                Transform target = CameraFollowObject.transform;
                float step = CameraMoveSpeed * Time.deltaTime;
                transform.position = Vector3.MoveTowards(transform.position,
target.position, step);
        }

}
```

Above is the camera-follow script, the start method is used as an initialisation so that any previous camera rotations are stored, so that it isn't assumed that the rotation of the camera is 0 or any other

standard, as this would make controlling the camera more difficult. The cursor is defaulted to the centre of the game window, once again for ease of use.

The update method is called each frame, in this script each frame I want to find the player's demand for the rotation of the camera. In my Unity project I implemented a controller interface with this script, Thus, moving the right joystick on a connected controller will be recorded by the RightStickHorizontal axis assigned in the project settings. I collect both mouse and controller inputs to get a final value for the magnitude by which the camera should be rotated. The rotation of the camera is then altered with respect to the frame rate, if the frame rate is lower, then the update method won't be able to collect as many values. So, the time taken for a frame to be executed can be used as a multiplier, such that low frame rates don't mean that the camera moves slowly. One issue with this is that the camera will look jagged at very low frame rates, although that is unlikely to happen, especially in this prototype.

The late update method determines the actions of the camera at the end of each frame. The CameraStepUpdate method is called immediately. This will determine the position of the player, then move cause the camera to move towards the player.

Camera Collision Script:

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

public class CameraCollision : MonoBehaviour
{

	public float minDistance = 0.2f;
	public float maxDistance = 5.0f;
	public float smooth = 10.0f;
	public Vector3 dollyDir;
	public Vector3 dollyDirAdjusted;
	public float distance; //


	void Awake()
	{
		dollyDir = transform.localPosition.normalized; // distance from camera
base, normalised to have a magnitude of 1
		distance = transform.localPosition.magnitude; // distance * dollyDir =
distance of the camera from the base
	}

	// Update is called once per frame
	void Update()
	{
		Vector3 desiredCameraPosition = transform.parent.TransformPoint(dollyDir
* maxDistance); // the desired position is the closest it can be to the camera base
when moving by the max distance
		RaycastHit hit;

		if (Physics.Linecast(transform.parent.position, desiredCameraPosition,
out hit))
```

```
            {
                  distance = Mathf.Clamp(hit.distance * 0.9f, minDistance,
maxDistance); // returns the distance from the camera to the hit object reduced by 10%
            }
            else
            {
                  distance = maxDistance;
            }
            transform.localPosition = Vector3.Lerp(transform.localPosition, dollyDir
* distance, Time.deltaTime * smooth);
      }

}
```

Above is the camera collision script. It is important to note that the camera system in this prototype is made up of two objects, a camera base, which uses the follow script, and the camera itself which uses the collision script. This script uses ray-casting to cast a ray from the camera base to where the camera is trying to move to, the raycast can then be used to determine if an object obstructs the path between the two positions. The camera then adjusts its position to avoid the obscuring object.

The fundamental of this system is that the base tries to get as close to the player as possible, the camera will then try to remain the 'maxDistance' away from the base.

Poisson Disc Sampling:

Poisson Disc Sampling is an algorithm that generates a list of points in any number of dimensions, where each point is no closer than a defined value 'r' to any other point, and where each point is within '2r' of at least one point. This model produces results that are reminiscent of those we see in nature. For example, trees exist in groups, but are never too close to each other, else their contest for nutrients and light hinders development. This makes the algorithm ideal for natural object placement in many cases. Such, I plan to prototype the algorithm to investigate the kind of distributions that it produces, and whether they would be suitable for my project. I have decided to prototype this system in Windows Forms, as it is relatively quick to set-up a simple UI, but fundamentally, Windows Forms allows me to draw pixels to the screen. I also feel that a visualisation of this algorithm may be suitable for a loading screen, and if I am to use it in my implementation, it is unlikely that it will be accompanied by a true visualisation. Therefore, my best opportunity to investigate this, is through prototyping.

PoissonDiscSampling.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace PoissonDiscSamplingWF
{
      public partial class Form1 : Form
```

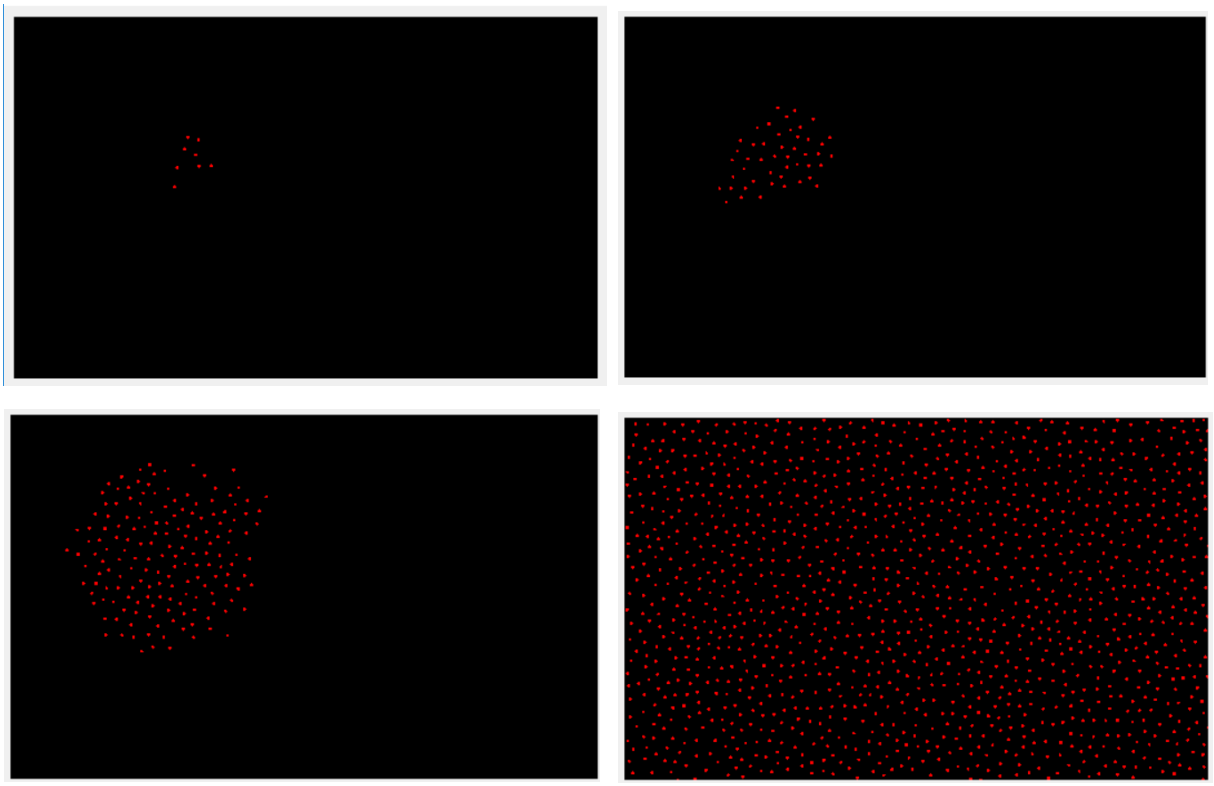
```
	{
		public static Random rand = new Random();
		public Settings settings;
		public int[,] grid;
		public List<Vector2> samples = new List<Vector2>();
		public List<Vector2> activeList = new List<Vector2>();
		public Form1()
		{
			InitializeComponent();
		}
		public void Run()
		{
			Setup();
			GeneratePoints();
			Draw();
		}
		public void Draw()
		{
			Graphics graphics = panel1.CreateGraphics();
			for (int i = 0; i < samples.Count; i++)
			{
				var t = Task.Delay(10);
				t.Wait();
				// plot each sample as a point 5 pixels wide
				graphics.FillEllipse(Brushes.Red, (float)samples[i].x,
(float)samples[i].y, 3, 3);
			}
		}
		public void GeneratePoints()
		{
			while (activeList.Count > 0)
			{
				int activeListIndex = rand.Next(0, activeList.Count);
				Vector2 activeSpawnPoint = activeList[activeListIndex];
				bool candidateAccepted = false;

				for (int i = 0; i < settings.getK(); i++)
				{
					// Generates a random position within the annulus
of r and 2r of the spawn point
					double angle = rand.NextDouble() * Math.PI * 2;
					Vector2 direction = new Vector2(Math.Cos(angle),
Math.Sin(angle));
					double magnitude = settings.getRadius() +
rand.NextDouble() * settings.getRadius();
					Vector2 offset = new Vector2(direction.getX() *
magnitude, direction.getY() * magnitude);
					Vector2 candidate = new
Vector2(activeSpawnPoint.getX() + offset.getX(), activeSpawnPoint.getY() +
offset.getY());

					// checks whether a point at the generated position
would be valid
					if (IsValid(candidate, settings.getWidth(),
settings.getHeight(), settings.getCellSize(), settings.getRadius(), samples, grid))
					{
						// if valid then the point is stored and
becomes a future spawn point
```

```
                                        samples.Add(candidate);
                                        activeList.Add(candidate);
                                        // stores point's index at the cell where the
point is is found
                                        grid[(int)(candidate.getX() /
settings.getCellSize()), (int)(candidate.getY() / settings.getCellSize())] =
samples.Count;
                                        candidateAccepted = true;
                                        break;
                                }
                        }
                        if (!candidateAccepted)
                        {
                                // no points could be spawned around the active
point
                                activeList.RemoveAt(activeListIndex);
                        }
                }
        }

        public bool IsValid(Vector2 candidate, int width, int height, double
cellSize, double r, List<Vector2> points, int[,] grid)
        {
                // Checks whether a point is within the sample area and whether
it is within the radius of any others
                if (candidate.getX() >= 0 && candidate.getX() < width &&
candidate.getY() >= 0 && candidate.getY() < height)
                {
                        // Finds the cell that the candidate is found in
                        // Meaning only surrounding 24 need be checked
                        int cellX = (int)(candidate.getX() / cellSize);
                        int cellY = (int)(candidate.getY() / cellSize);
                        // Min and Max prevent errors when candidate is found on
the edge
                        int startX = Math.Max(0, cellX - 2);
                        int startY = Math.Max(0, cellY - 2);
                        int endX = Math.Min(cellX + 2, grid.GetLength(0) - 1);
                        int endY = Math.Min(cellY + 2, grid.GetLength(1) - 1);

                        for (int x = startX; x <= endX; x++)
                        {
                                for (int y = startY; y <= endY; y++)
                                {
                                        int pointIndex = grid[x, y] - 1;
                                        if (pointIndex != -1)
                                        {
                                                // -1 marks an empty cell as indexing
will begin at 0
                                                double sqrDstX = (candidate.getX() -
points[pointIndex].getX()) * (candidate.getX() - points[pointIndex].getX());
                                                double sqrDstY = (candidate.getY() -
points[pointIndex].getY()) * (candidate.getY() - points[pointIndex].getY());
                                                double sqrDst = sqrDstX + sqrDstY;
                                                if (sqrDst < r * r)
                                                {
                                                        // candidate is within the
radius of an existing point
```

```
                                                        // candidate is therefore
rejected
                                                        return false;
                                                }
                                        }
                                }
                        }
                        // candidate placement is valid and is accepted
                        return true;
                }
                // candidate wasn't spawned within the bounds of the sample area,
therefore rejected
                return false;
        }

        public void Setup()
        {
                settings = new Settings(30, 2, 9, 700, 400);
                this.Size = new Size(settings.getWidth(), settings.getHeight());
                grid = new int[(int)Math.Ceiling(settings.getWidth() /
settings.getCellSize()), (int)Math.Ceiling(settings.getHeight() /
settings.getCellSize())];
                double x0 = rand.Next(1, settings.getWidth());
                double y0 = rand.Next(1, settings.getHeight());
                activeList.Add(new Vector2(x0, y0));
        }
        public class Settings
        {
                private int k { get; set; }
                private int n { get; set; }
                private double r { get; set; }
                private double sqrt2 = 1.41421356237;
                private int width;
                private int height;
                private double cellSize { get; set; }
                public Settings(int k, int n, float r, int width, int height)
                {
                        this.k = k;
                        this.n = n;
                        this.r = r;
                        this.width = width;
                        this.height = height;
                        this.cellSize = r / sqrt2;
                }
                public int getWidth()
                {
                        return this.width;
                }
                public int getHeight()
                {
                        return this.height;
                }
                public double getCellSize()
                {
                        return this.cellSize;
                }
                public double getRadius()
                {
```

```
                        return this.r;
                }
                public int getK()
                {
                        return this.k;
                }
        }
        public class Vector2
        {
                public double x;
                public double y;
                public Vector2(double x, double y)
                {
                        this.x = x;
                        this.y = y;
                }
                public double getX()
                {
                        return x;
                }
                public double getY()
                {
                        return y;
                }
                public void setX(double x)
                {
                        this.x = x;
                }
                public void setY(double y)
                {
                        this.y = y;
                }
        }

        private void button1_Click(object sender, EventArgs e)
        {
                Run();
        }
    }

}
```

Above is the script I have created to visualise the Poisson Disc Sampling algorithm. The output of this program is a visualisation of the points in a 2D space. Despite being a prototype test, I wanted to keep the program object-oriented as that is how I would try to implement the algorithm in my solution.



Like the A* algorithm, the PDS algorithm has a frontier. This frontier consists of all points which should be considered as a spawn point in the future. When we consider the spawn position of a new point, we can presume that the point is most likely to spawn on the edge of the cluster. This is because points that attempt to spawn within the cluster are more likely to fall within 'r' of another point than those on the perimeter. Furthermore, if it were possible for a point to spawn within the cluster, it is likely that it would've already done so from a previous spawn point and thus further reduced the space in the cluster. Such, when viewing the algorithm, it appears as though the cluster radiates outwards towards the edges. To view this, I added a slight delay before drawing each point.

The nature of the algorithm also allows the value of 'r' to be altered to change the separation of samples. This is of course desirable for my purpose, as different objects lend themeselves to certain separations. Figure 1 shows a completed simulation where the radius has been doubled in comparison to the initial test.

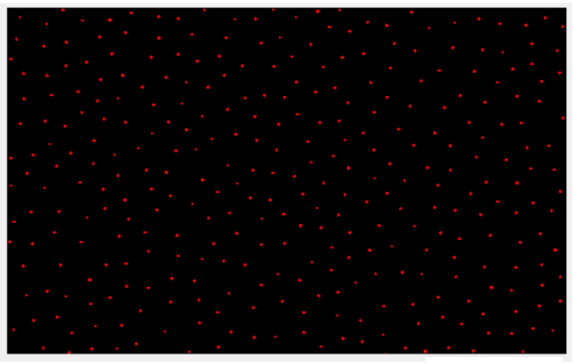


Figure 1

DATA VOLUME:

From my research and prototyping I confirmed my previous assumption that my program would have to handle a large volume of data. The majority of this data would be involved with creating the terrain mesh and populating it with thousands of objects. As the map is randomly generated, I can't create values and save them in an external location when I am trying to initially create an island. Therefore, all of the values will need to be handled by the game engine and scripts at the time when an island is created. Thus, I confirmed that it would be unlikely for me to be able port my game to anything but a powerful PC or console.

If I achieve allowing players to revisit previous islands, I made need a database to store information about the previous island. The volume of this data will likely be very similar to that required in creating the island, as I feel that the only way for terrain to be recreated is for all of its values to be stored. Such, I could also need a database to handle a large amount of data as well as my game engine.

OBJECTIVES:

1. Procedurally generate random, traversable and characteristic terrain using a 3D noise function with applied variables.

    a) Overlay noise in multiple octaves to increase the detail of the terrain.
    b) Use lacunarity and persistence to alter the frequency and amplitude between octaves.
    c) Use different styles of terrain to ensure that the 'world' is suitably varied.
    d) Create the terrain as an island using the influence of a falloff map.
    e) Vary the size of map that can be created by joining terrain areas together.
    f) Using the noise values as vertices, create a mesh with an accurate terrain collider.
    g) Join terrain areas together such that there is no visible seam.

2. Randomly spawn natural biomes onto the terrain in suitable and realistic locations.

    a) Use terrain height to find appropriate locations for natural game objects.
    b) Spawn natural game objects with characteristic separation or frequency.

3. Create a detection system for the enemies such that they can locate and approach the player.

    a) While the player is undetected, the enemy should patrol in random directions.
    b) Use a visibility algorithm such that the player can only be detected by an enemy when they're within a certain radius of the enemy, and within their line of sight.
    c) [Extension] The player's activity in the proximity of the enemy, such as making a noise can influence the detection algorithm.

d) If the player alerts the enemy or is detected, the enemy will move towards the player as long as the player remains detected.
e) Objects in the world will obscure the enemies' view of the player, enabling the player to become undetected.

4. Implement an interactive 3rd person camera system.

a) The player should be the focus of the camera during standard gameplay, with the player having the ability to orbit the camera around themselves at any time.
b) The camera should follow the path it is led by the player, but automatically move around obstacles that would obscure the view of the player.
c) The camera should remain a fixed distance away from the player unless it is forced to manoeuvre.

5. Spawn a range of collectable items in the world to assist the player in traversal or combat.

a) Spawn an object that will reveal the player's location in relation to the entire map.
b) Spawn an object that will reveal the position of all enemies on the map.
c) Spawn an object that will add a compass to the player's navigation system.
d) Spawn an object that will reveal a selected path to the goal on the player's map.
di) The path should seek to avoid steep ascent or descent.
dii) The path should seek to avoid entering into close proximity of enemies, only however, once the player has revealed the enemies.
e) Spawn an object that must be collected for the player to win the game.
f) [Extension] Spawn objects that enable the player to engage in combat.
g) [Extension] Spawn objects that enable the player to engage in combat.

6. Create a simplistic and informative UI to display game information.

a) Create a player HUD to display frame relevant information during the game.
ai) Display the player's current health at all times during the game.
aii) The player should lose the game when their health falls to 0.
aiii) Display a minimap showing nearby ground when the game is in an active state.
b) [Extension] The player is able to disable as much of the HUD as they choose.
c) Develop an inventory menu where the player has a large display of the entire map.

7. [Extension] Create an interactive and skill-based combat system.

a) The combat system is developed using a variety of animations, influenced by the way in which the player attacks.
b) The timing of the player's actions during combat will have a greater effect, such as dodging attacks or blocking.

8. [Extension] Allow players to revisit previous islands.

a) When an island is created, all of its attributes are stored such that it can be exactly recreated again if the player chooses to return to the island.
b) When an island is revisited, items and enemies that weren't previously found on the island will spawn, giving a purpose to returning to an island.

Design

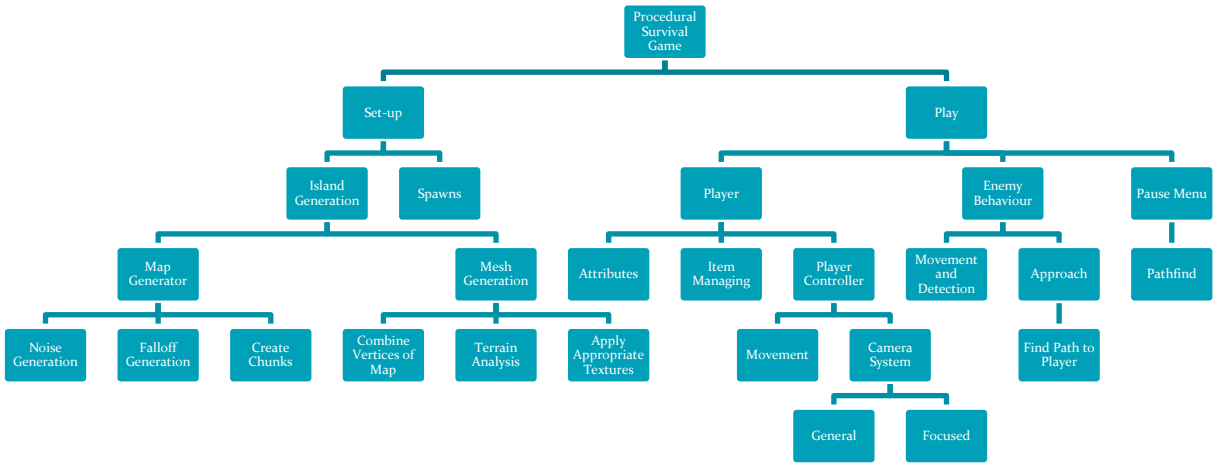
OVERVIEW

I plan to implement my project using the Unity game engine, where I will be able to use C# as my scripting language. Unity will be helpful in the creation of the game UI as well as the aesthetic of the procedural generation. The procedural generation will occur prior to the game, but I may consider implementing a loading screen should the algorithm take too long. Along with this I expect that I will need to implement a rendering distance threshold to prevent the game slowing down dramatically on large maps with a plethora of objects. The project will consist of two scenes, a menu scene and a game scene, where all the gameplay will occur.

VISUALISATION

Hierarchy:

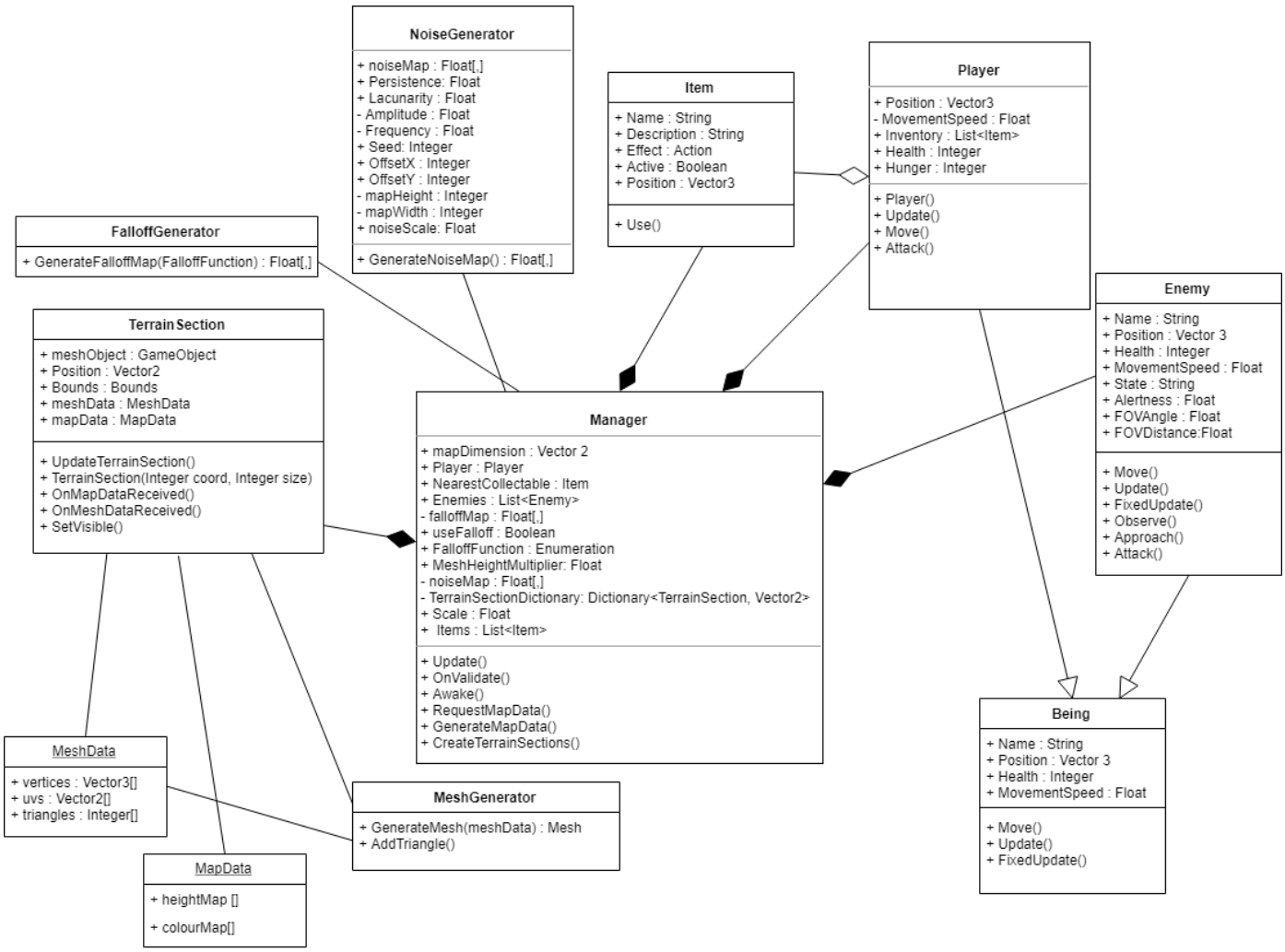
This hierarchy diagram shows the procedures that are required through the progression of the game and the order in which they occur. There is a defined split between the set-up and runtime procedures that make up the program.



The game begins with a set-up, where the island is generated by manipulating the Perlin Noise through multiple stages. A mesh is then generated using the height values from the noise map as vertices in 3D space. The terrain is then analysed, to determine the type of terrain that it would best represent, meaning that an appropriate texture can be applied. Once the island has been created and it has been populated with enemies, collectables and wildlife, then the game can be played. The Player will have several attributes relating to their status, they will have a collection of items that make up their

inventory, and the player will have a movement system. The enemies will also have movement systems, that will be changed by the detection algorithms. When accessing the pause menu, the system may need to execute the pathfinding procedure, given that the player has found the appropriate collectables. Some of these procedures may not be implemented as they are challenge objectives, furthermore the order of some of the operations may also change in implementation.

Class Definitions (UML Diagram)



The UML diagram represents all of the classes that are required in the preparation or execution of the game. The abundance of public variables may seem to be poor design, however public variables are displayed in the Unity inspector, allowing their values to be input by an automatically created UI as well as be validated. Thus, most of fields are public, however in cases where I am certain that the value would not benefit from the public functionality, I have declared the attribute privately.

The central Manager class begins by creating all instances of the non-static classes that it is associated with. This class plays an important role in the game set-up as it ensures that all of the procedures are called in the correct order while managing the different threads. Having created all instances of the non-static classes that the manager is associated with, its main role becomes managing the gameplay and interactions between classes.

All of the living creatures in the scene will inherit from the being class, this contains all of the basic attributes that apply to all organisms. The Player class inherits from this, having only one instance, stored in the static Manager class. The Player class overrides methods from the Being class when appropriate, as the player will likely have more operations in the Update methods than other beings in the world. Within the Player class there is a list of items that make up the player's inventory. A similar list is stored in the Manager class which refers to all the items on the map, along with an attribute that stores the closest collectable such that a path can be created from the player's location to the item. Enemies differ from the player slightly as they store attributes related to the detection of and approach to the player. All beings other than the player will be treated as enemies.

The noise generator and falloff generator both have a simple main function to produce a 2D float array to return to the manager such that an overall height-map can be created. The classes themselves are relatively bare, however it is possible that implementation may require a greater volume of data to be stored within the classes. The classes are both static as only one instance need exist, and the classes also need only be used once.

The Terrain Section class is used to store all the data about a certain chunk of land that makes up the world. By splitting the game-world into multiple sections, it means that multiple meshes can be created rather than a singular mesh. This provides a greater level of detail in the terrain without compromising size, in addition to allowing the map to be easily scalable in size. Each terrain section is stored in a dictionary that relates each section to a 2D coordinate in game-space. This is the only location where the terrain sections are stored, and they don't exist outside of the container. Thus, to create each terrain section, the manager class provides the correct section of the height-map as a parameter for the constructor using the MapData structure.

The use of the MapData and MeshData structures allows all the summary information regarding each to be transferred and stored in a concise manner. Such, the mesh for each terrain section can be created from only the three fields in the structure. Implementation may highlight some flaws with the proposed design, however the approach means that the program can be developed with a modular approach making it easier to locate and fix any errors.

DATA

I have created a data dictionary to show the variables that are used in gameplay, and the classes that they are associated with. Along with that I have given a brief description of the purpose and use and each variable as well as its data type. It is possible that when approaching the challenge objectives of my project, some of the variables may instead be stored in a database. This could include the seed for a certain map, and the noise variables, should they be altered between generations.

Data Dictionary: Player Class

Variable	Description	Data Type
Position	Represents the player's location in 3D game-space.	Vector 3
Health	Stores the player's current health (max 100). This can be reduced by enemies and hunger and be replenished using items.	Integer
MovementSpeed	Represents the speed at which the player moves in relation to other beings. Can be altered by the use of items.	Float
Collectables	Groups the collectable items that the player can gather during the game.	Dictionary<String, Item>

Data Dictionary: Noise Generator Class

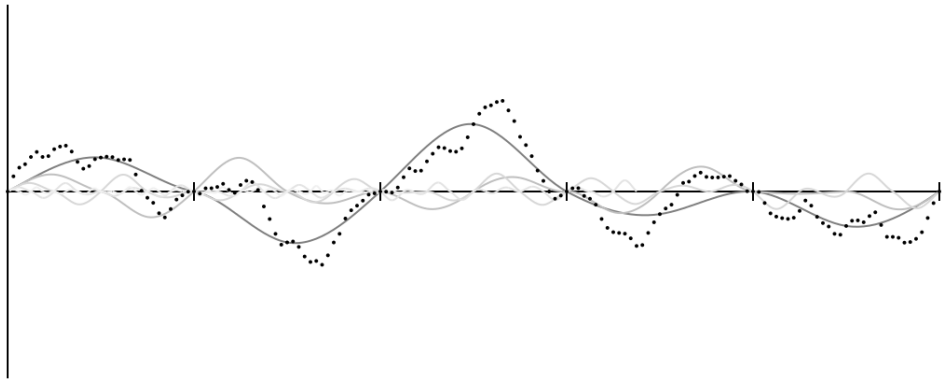
Variable	Description	Data Type
Amplitude	Represents the amplitude to be applied to noise values, thus acting as a scale factor in the y-axis.	Float
Frequency	Represents the frequency at which the noise is sampled, a smaller value will mean a more spread out section of noise is used.	Float
Lacunarity	Constant value used to influence the frequency of the noise, it is a multiplier that determines how quickly frequency increases with each successive octave. Greater than 1 to increase the frequency between octaves.	Float
Persistence	Constant value used to influence the amplitude of the noise, it is a multiplier that determines how quickly amplitude diminishes with each successive octave. Between 0 and 1 such that amplitude decreases between octaves.	Float
MapHeight	Holds the number of y coordinates that will be held in the 2D noise map array.	Integer
MapWidth	Holds the number of x coordinates that will be held in the 2D noise map array.	Integer

NoiseMap[,]	Holds the height value of the mesh for each (x,z) coordinate of noise. Normalised between 0 and 1.	Float
Seed	Used to create a random number that determines the position at which the noise is sampled. By storing the value, the same terrain can be generated again.	Integer
OffsetX	Represents the x coordinate at which the noise is first sampled.	Integer
OffsetY	Represents the y coordinate at which the noise is first sampled.	Integer
MinLocalNoiseHeight	Stores the minimum height value in the noise map. Can be used to calculate height range, to compare each section of noise with others to determine terrain type.	Float
MaxLocalNoiseHeight	Stores the maximum height value in the noise map. Can be used to calculate height range, compare each section of noise with others to determine terrain type.	Float

ALGORITHMS

Noise-Map Generation:

We generate the noise-map by iterating through two loops, allowing each coordinate in a 2D array to be assigned a value from Perlin Noise. As we want to create realistic variation in the terrain, we shall layer multiple noise functions over each other. However, each successive noise function should be different in frequency to the previous, else it will have the effect of simply amplifying the previous function. The amplitude should also decrease with each successive function to reduce its effect on the terrain, allowing more subtle variation to be achieved. Diagram 1 shows an example of three octaves being layered where the multiplier of frequency (lacunarity) is 2.



The variables sampleX and sampleY are used to allow different scopes of noise to be used, to improve variation in the noise-map. The minimum and maximum noise-height for the generated section are

stored so that the noise-map can be linearly interpolated between the two values. Following each octave, the amplitude and frequency are altered using the persistence and lacunarity values respectively.

```
FOR y <- 0 TO mapHeight – 1
        FOR x <- 0 TO mapWidth -1
                noiseHeight <- 0
                amplitude <- 1
                frequency <- 1
                FOR i <- 0 TO octavesNo -1
                        sampleX <- (x – (mapWidth/2f + octaveOffset[i].x)) / (scale * frequency)
                        sampleY <- (y – (mapHeight/2f + octaveOffset[i].y)) / (scale * frequency)
                        perlinValue <- Math.PerlinNoise(sampleX, sampleY) * 2 – 1
                        noiseHeight += (perlinValue * amplitude)
                        amplitude *= persistence
                        frequency *= lacunarity
                ENDFOR

                IF noiseHeight > maxLocalNoiseHeight THEN
                        maxLocalNoiseHeight <- noiseHeight
                ENDIF
                IF noiseHeight < minLocalNoiseHeight THEN
                        minLocalNoiseHeight <- noiseHeight
                ENDIF
        ENDFOR
ENDFOR

FOR y <- 0 TO mapHeight -1
        FOR x <- 0 TO mapWidth – 1
                noiseMap[x,y] <- (noiseMap[x,y] – minLocalNoiseHeight)/(maxLocalNoiseHeight –
minLocalNoiseHeight)
        ENDFOR
ENDFOR

RETURN noiseMap
```

Mesh Generation:

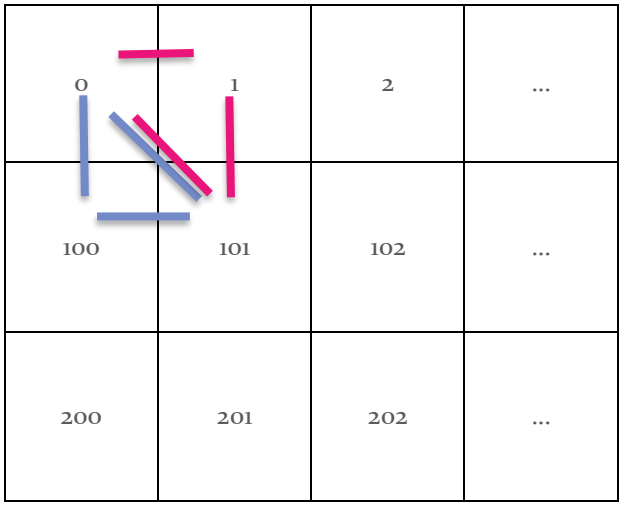


Diagram 1

When manually creating a mesh, we need an 1D array of vertices along with a triangles array that illustrates how the vertices should be connected. However, as we are using each position in our 2D array as a vertex in our mesh, we must transfer our 2D array into a 1D array. Consider a noise-map with width 100, where each coordinate forms a vertex in our mesh. Diagram 1 shows a visualisation of our 2D array, where the number at each vertex represents the index that we can give that coordinate so that we can transfer our 2D array into a 1D array of vertices. In order to create a mesh, each position must store a Vector3. In terms of our map, we can consider thisVector3 to store the coordinate's position on the x,z plane along with its height in the y direction, in essence the vertex's position in 3D space. The triangles drawn on Diagram 1 show the way in which we connect the vertices to create the triangles needed for the mesh. Our triangles array must also be 1D, with each consecutive set of 3 integers illustrating the index - in the vertices array - of the vertices that make up that triangle. We shall construct each triangle in the clockwise direction, thus ,to define our first triangle we would add to the triangles array 0, 101, 100. To complete the other half, we would add the integers 101, 0, 1. We can continue this process to connect every square of 4 vertices to each other by adding two triangles to the array each time.

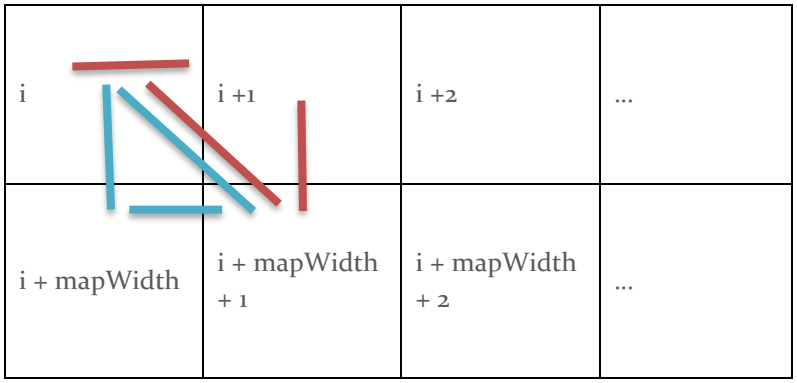


Diagram 2: Where 'i' is the index of a For loop

However, we want to centre the mesh about 0,0, this means that the width and height of the mesh must both be odd such that we have a centre point with an equal number of vertices on all sides. Therefore, when storing the x and z positions of each vertex, we must do so in relation to the leftmost x value, and the upmost z value. These will be given by (map width – 1) / -2 and (map height – 1) / 2 respectively. Diagram 2 illustrates how we can consider the vertex index's when iterating through loops. Note that i will increase by 1 following each set of two triangles, thus the expression for the four vertices will always be the same. The following pseudocode demonstrates this idea and how we use it to populate the vertices and triangle arrays. The AddTriangle() method simply adds the parameters to the end of triangle array in consecutive order.

```
topLeftX <- (mapWidth – 1) / -2
topLeftZ <- (mapHeight – 1) / 2


verticesPerLine <- (mapWidth – 1)

vertexIndex <- 0

FOR y <- 0 TO mapHeight – 1
        FOR x <- 0 TO mapWidth – 1
                vertices[vertexIndex] <- Vector3(topLeftX + x, heightmap[x,y] * heightMultiplier,
topLeftZ – y)
                uvs[vertexIndex] <- Vector2(x / mapWidth, y / mapHeight)
                IF (x < mapWidth -1 AND y < mapHeight – 1 THEN
                        AddTriangle(vertexIndex, vertexIndex + verticesPerLine + 1, vertexIndex +
verticesPerLine)
                        AddTriangle(vertexIndex + verticesPerLine + 1, vertexIndex, vertexIndex +
1)
                ENDIF
                vertexIndex +=1
        ENDFOR
ENDFOR
```

Falloff-Map Generation:

When generating the falloff-map we want to create a 2D array that we can then apply directly onto an equally sized noise-map. Therefore, we need a function that will output a map where the values increase outwards from the middle, however the ascent shouldn't be linear, as otherwise the map would often have its highest point in the centre and wouldn't be particularly varied. Thus, the function should output 0 when the input is close to 0, and a value close to one, when the input is 1. When finding an input for the function we can take the modulus of the x and y coordinates of the point that we are mapping in relation to the centre, and then use the greatest of the two as the input for our falloff-function, this we give the falloff-map an almost quadrilateral shape. Alternatively, we could take the magnitude of (x + y) for a more cyclic shape. The two equations below represent falloff functions that I experimented with in prototyping, this is because both map 0 to 0, and practically 1 to 1.

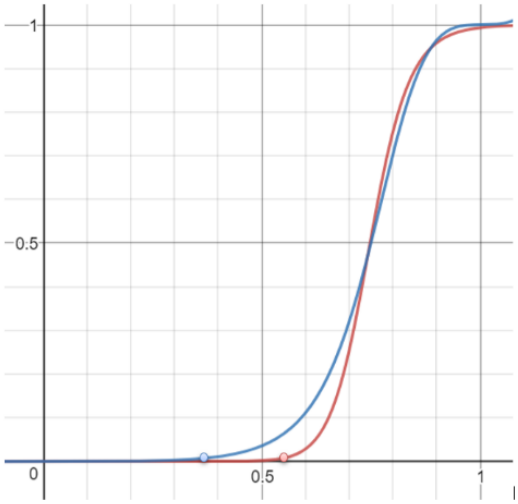
$$f(x) = \frac{e^{6x-1}}{e^{6x-1} + x^c}$$

Equation 1

$$f(x) = \frac{x^a}{x^a + (b - bx)^a}$$

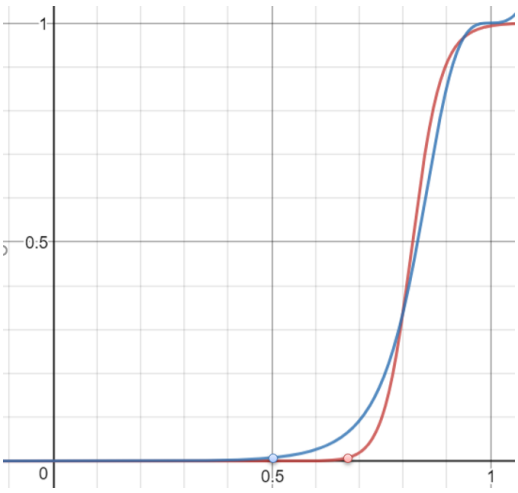
Equation 2

Where f(x) is the value of the falloff-map at the input coordinate maximum.



Graph 1: Equation 1 , Equation 2

c = -12, a = 3, b = 3



Graph 2: Equation 1 , Equation 2

c = -20, a = 3, b = 5

Equation 1 is based off the sigmoid function as I think that its characteristic shape could be modified to create an effective falloff map. I reached the final function after some experimentation on graphing software to ensure that an input of < 0.5 would be less than the 'activation input'[1] and result in no falloff output. I found Equation 2 online when looking for suitable functions to use in falloff-map generation. The function is very similar in shape to Equation 1, however the 2 constants allow for increased variation.

[1] The minimum input before the gradient increase of the falloff function becomes significant, shown as a circular marker.

When comparing the functions we can see that with the above constants, Equation 1 has a higher 'activation input' than Equation 2, thus giving Equation 1 a greater average gradient between the 'activation input' and 1. We can also see that Equation 1 has an almost linear gradient for the majority of its ascent, this will result in a more constant increase in the values of the falloff-map is a radiates outwards.

From looking at the variation between Graphs 1 and 2, we can observe that by increasing the magnitude of the negative exponent 'c', the most prominent change to Equation 1 is that the 'activation input' is increased, thus giving the falloff-map a greater gradient and a greater distance before the falloff map takes effect. By increasing the value of 'b', similarly the 'activation input' increases, as does the average gradient.

Now we can consider what this means for the resulting terrain. As the dimensions of the map increases the 'activation input' must in turn increase, else a threshold will be reached where the outer terrain sections are simply water, which would be a waste of processing. Thus, we must increase the 'activation input' with the map dimensions. The below equations express suitable values for 'b' and 'c' that allow the falloff-map to scale appropriately as the map grows larger. 'a' is kept constant as it allows Equation 2 to keep its characteristic shape.

$$\boldsymbol{b = Average\ Map\ Dimension} \qquad \boldsymbol{c = \frac{Average\ Map\ Dimension}{2} \times -4}$$

We also noted that Equation 1 has a greater average gradient between its 'activation input' and 1, this means that when the falloff-map is applied to the Perlin Noise, the resulting terrain height will decrease at a greater rate when Equation 1 is applied than if Equation 2 were used. The large disparity between the resulting 'activation input' of the two equations will mean that when Equation 1 is applied, the height of the terrain will start decreasing from closer to the centre of the map. Another noteworthy difference is the much flatter plateau of Equation 2, this will result in areas close to the shoreline all being of a very low height, whereas with Equation 2, those areas could easily have a sizeable difference in height.

The following pseudocode shows an implementation of the ideas discussed. We must determine the section of the falloff-map that has been requested, and then create it using the chosen function. The value of the Enumeration 'falloffFunction' can easily be altered in the Unity inspector.

```
SUBROUTINE GenerateFalloffMap(Int terrainSectionSize )
FOR i <- 0 TO terrainSectionSize * mapDimensions.x
        FOR j <- 0 TO terrainSectionSize * mapDimensions.y
            x <- i / (terrainSectionSize * mapDimension.y) * 2 – 1
            y <- i / (terrainSectionSize * mapDimension.x) * 2 – 1
            m <-  Mathf.Max( |x| , |y| )
            falloffMap[i , j] <- Evaluate(m, mapDimension, falloffFunction)
```

```
            ENDFOR
ENDFOR
    RETURN falloffMap[i , j]
END SUB

SUBROUTINE Evaluate(Float input, Vector2 mapDimension, Enumeration falloffFunction)
    IF falloffFunction = falloffFunction.Sigmoid THEN
        a <- -4 * (mapDimension.x + mapDimension.y) / 2
        RETURN ( e ^ ( 6 * input – 1) / e ^ ( 6 * input – 1) + input ^ a
    ELSE
        a <- 3
        b <- (mapDimension.x + mapDimension.y)/2
        RETURN   ( input ^ a ) / ( input ^ a + ( b – b * input ) ^ a)
    ENDIF
ENDSUB
```

Object Placement:

When spawning objects on the world we need to consider the characteristics of the object being spawned, as this will likely tell us information about the number and separation of the objects as well as the kind of locations at which they spawn. In a 2007 paper Robert Bridson suggests a method that generates points within an area such that each point is greater than a minimum distance from all others, but within a maximum distance of at least one point, this is known as Poisson Disc Sampling. By following the suggestion in the paper, it will be possible to spawn trees within a certain range of each other and within a certain height range.

Let's consider how the algorithm works, first we need to introduce the key constants that define the outcome of the algorithm. In short, the algorithm generates points such that each point is within '2r' of at least one point, and no closer than 'r' to any other.

$$r = Minimum\ Distance\ Between\ Two\ Samples$$

$$n = Number\ of\ Dimensions$$

$$k = Number\ of\ Samples\ to\ Test\ Before\ Rejection$$

$$c = Dimension\ of\ a\ Cell\ in\ the\ Grid$$

The paper suggests an implementation of the algorithm in multiple dimensions, however we need only focus on two, in the case of the trees, thus our n = 2. Our grid will have dimensions equal to area that we might want to spawn objects in, thus in most cases it will be the global dimensions of the map.

We want each cell in our grid to contain no more than one sample, therefore we want the diagonal of each cell to be r:

$$\mathbf{c} = \frac{\mathbf{r}}{\sqrt{\mathbf{n}}}$$

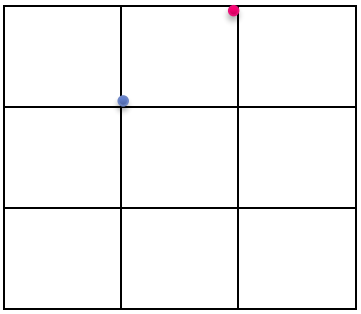


Diagram 1

Diagram 1 helps us to understand why this is true. In Diagram 1 the red sample has been spawned in the upper-right corner of a cell. The blue point shows a point that is the greatest distance from the red sample, but still within the cell, this is of course the far corner of the same cell. As these points are on the cusp of being accepted, the distance between them must be r, as if the blue point were to move any further away, it would be in a different cell. We also know that our cells have equal width and height, so we use 'c' to describe both. Thus, from Pythagoras' Theorem we can formulate an equation, where all that must be done to find c, is a substitution using n and a simple rearrangement.

$$r^2 = 2 \times c^2$$

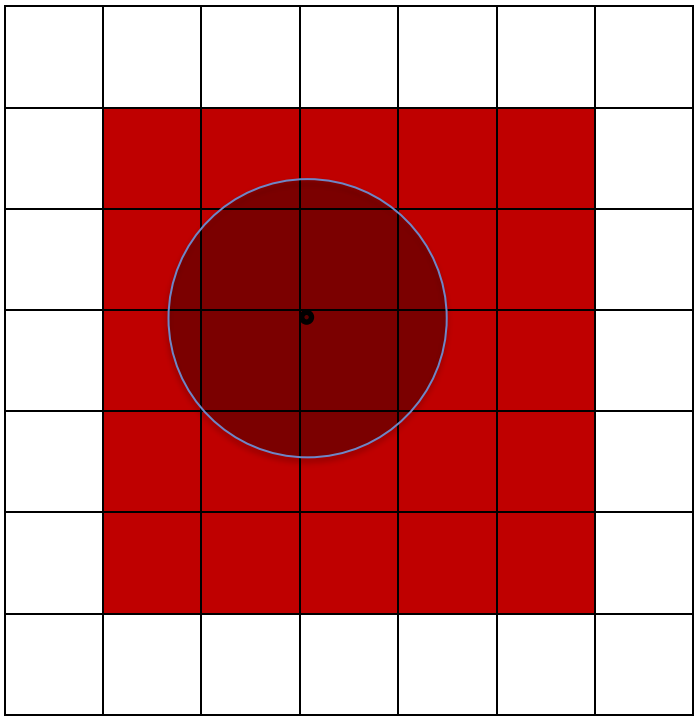


Diagram 2

We can store this grid using a 2D array, where if a cell contains a sample, then the grid stores the index of the sample. We can now spawn our first sample, this can be at random position within our sample area. We will add this point to two different lists, the first that stores each point - we can call it our 'points' list - and a second, called the 'active list', that stores all the points to be used as spawn points. With this set-up, we can begin the algorithm.

While the active list isn't empty, we select a random point from it to be our spawn point. Then we generate up to 'k' points within the annulus of r and 2r of the spawn point. We then must check each of these points to see whether they fall within a distance of r of any of the samples in the 'points' list, as this will render this new point invalid. If the point isn't deemed invalid, then we can add it to both lists, and store its index in the grid. If k iterations occur and no valid points are found then the spawn point is removed from the active list, so that it won't be used again. Fortunately, due to the nature of our grid system, when checking the validity of a new point, we don't need to check it against all the previous points.

From Diagram 2 we can see that no matter where the point is placed within the central cell in the 7x7 grid, it isn't possible for the radius of the circle to extend beyond a 5x5 grid centred on the middle cell, such is the nature of our cell size definition. Thus, when validating a point, we need only check the points in the area. This helps the algorithm to perform efficiently even when the sample area grows very large.

The pseudocode below shows a method of generating a list of points within a desired area, using the Poisson Disc Sampling method. The first subroutine generates a list of points, using the second to validate each candidate before selecting it as a point. The pseudocode follows from the ideas discussed and those in Bridson's paper. A random angle and distance are generated for each candidate in order to place points in the annulus of the spawn point. The subroutine returns a points array that could then be used to place objects at those points.

```
SUBROUTINE GeneratePoints(Float r, Vector2 sampleRegionSize, Int k)

        n <- 2;
        c <- r / Mathf.Sqrt(n)
        iniPoint <- Vector2( RANDOM_FLOAT(1, sampleRegionSize.x – 1), RANDOM_FLOAT(1,
sampleRegionSize.y -1)
        points.Add( iniPoint )
        activeList.Add( iniPoint )
        grid[iniPoint.x / c , iniPoint / c] <- 1

        WHILE activeList.Count > 0
                activeListIndex <- RANDOM_INT(0, activeList.Count)
                activeSpawnPoint <-activeList[activeListIndex]
                candidateAccepted <- FALSE
                FOR i <- 0 TO k – 1
                        angle <- RANDOM_FLOAT(0, 2 * π)
                        direction <- Vector2(Cosine(angle), Sine(angle))
                        magnitude <- RANDOM_FLOAT(r, 2 * r)
                        offset <- direction * magnitude
                        candidate <- activeSpawnPoint + offset
                        IF IsValid() = TRUE THEN
                                points.Add(candidate)
                                activeList.Add(candidate
                                grid[candidate.x / c, candidate.y / c] <- points.Count
                                candidateAccepted <- TRUE
                                BREAK
                        ENDIF
                ENDFOR
                IF candidateAccepted = FALSE THEN
                        activeList[activeListIndex].Remove
```

```
            ENDIF
        ENDWHILE

        RETURN points
ENDSUB
```

The IsValid() function is used to check whether the current candidate falls within the radius of any previously established points. It uses the idea that only a 5x5 grid surrounding the cell need be checked, this allows the algorithm to operate at O(n). The 'start' and 'end' variables are validated using the Max and Min functions, which return the highest and smallest of two values respectively. This is important for when a cell is near the edge, because there may not exist a cell that is two to the left of the candidate's cell, and thus checking such a cell would result in an error. The paper suggests marking each cell that doesn't contain a sample with -1, also the first point will have a value of 1 in the grid, and indexing lists begins at 0, thus 1 is subtracted from whatever is stored in the grid to give a pointIndex, where -1 marks an empty cell.

```
SUBROUTINE IsValid(Vector2 candidate, Float c, Vector2 sampleRegionSize, Float r, Int grid[,])

        IF candidate.x >= 0 AND candidate.x < sampleRegionSize AND candidate.y >= 0 AND
candidate.y < sampleRegionSize

                cellX <- candidate.x / c
                cellY <- candidate.y / c
                startX <- Mathf.Max(0, cellX – 2)
                startY <- Mathf.Max(0, cellY – 2)
                endX <- Mathf.Min(cellX + 2, grid.LengthX – 1)
                endY <- Mathf.Min(cellY + 2, grid.LengthY – 1)

                FOR x <- startX TO endX
                        FOR y <- startY TO endY
                                pointIndex <- grid[x,y] – 1
                                IF pointIndex != -1 THEN
                                        sqrDistance = (candidate – points[pointIndex]).Sqr
                                        IF sqrDistance < r * r THEN
                                                RETURN FALSE
                                        ENDIF
                                ENDIF
                        ENDFOR
                ENDFOR

                RETURN TRUE
```

ENDIF

RETURN FALSE

ENDSUB

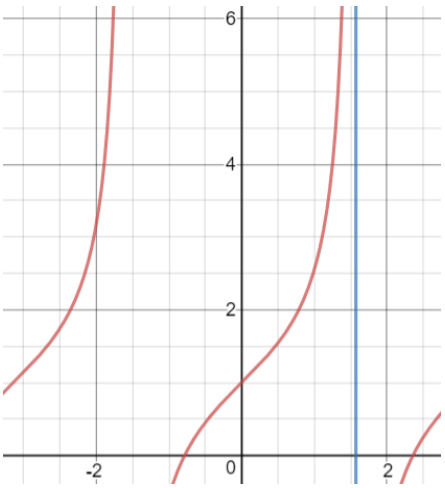
Pathfinding:

Due to the nature of our noise map, our terrain should be quite varied in height. The standard Third Person Controller in Unity deters characters from walking up or down steep inclines. Thus, when suggesting the quickest path to a collectable for our player, we want to avoid travelling through hills. For this we must define a cost function that will determine the movement cost between two nodes on the map, based on the 2D distance between them, and the angle that they make with each other and the z-plane.

$$f(x) = \tan(x) + 1 \qquad f(x) = \frac{\pi}{2}$$

Equation 1 *Equation 2*

Graph1, Equation 1, Equation 2

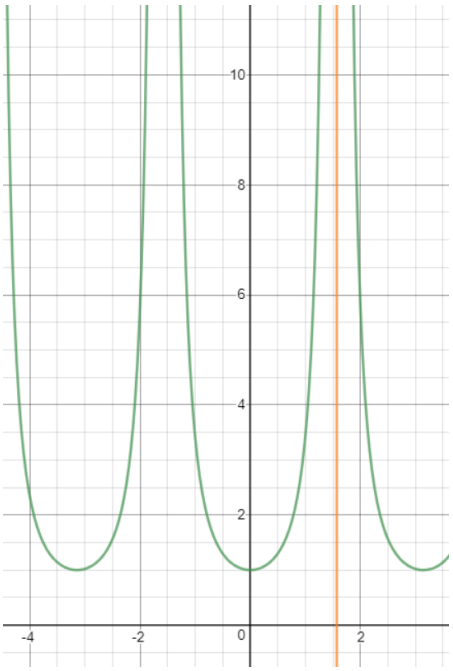


Graph 1 shows an illustration of Equation 1, which I have chosen to use in the cost function. The nature of tan makes it suitable for such a purpose, as the size of the angle increase, the value of tan increases. However, this doesn't occur at a linear rate, which is ideal, as the cost will rapidly increase as the angle increases. We can find this rate by differentiating our cost function, shown in Equation 3 and Graph 3.

$$\frac{dy}{dx} = sec^2(x)$$

Equation 3

From Graph 3 we can see that the gradient of our cost function will begin at 1, and then rapidly increase as it approaches π/2 (orange line). This should lead our algorithm to avoid steep ascents unless necessary. The Third Person Character struggles to move downhill, but not as much. To reflect this, we will use Equation 1, but we will half the tan value. This will half the rate at which the cost increases.



Graph 3

To improve the speed of our algorithm we use a heuristic, this is an estimate of distance required to reach the goal. As the A* algorithm is based off the Breath First Search, without the heuristic, the frontier would move in all directions until the goal

was reached. Although this may find a better path, it will take far longer than initially moving in the direction of the goal. During implementation, it will be important to find a balanced heuristic.

We also want our enemies to influence the path that is generated, such that our path advises players on how to reach the goal while avoiding enemies. To do this, we will take the current position of all of the enemies, then at each node we can increase the cost by a function of the enemies' proximity.

The below pseudocode illustrates a simple subroutine that determines the movement cost between two neighbouring nodes. Storing the value of 'tan' rather than finding an angle and performing trigonometry on it should improve the execution time of the subroutine which can make a large difference when it is being called very frequently.

```
SUBROUTINE MoveCost(Coordinate from, Coordinate to)

        distance = |from.x – to.x| + |from.y – to.y|
        tan = (to.height – from.height) / distance

        IF tan >= 0 THEN
                Cost <- 20 * tan + 1
        ELSE
                Cost <- -10 * tan + 1
        ENDIF

        RETURN cost * distance
ENDSUB
```

As I plan on using the A* algorithm, we need to heuristic to influence the order of nodes that we visit. To do this we can use a priority queue, this will sort the nodes by their priority, meaning that the nodes will be accessed in order of priority. We can determine this priority using the heuristic and use a binary insertion to add new nodes to the queue. I will implement a class called Coordinate, each Coordinate will be a node in the graph, allowing me to store information about each node within the class, as well as create methods to assist the algorithm.

Pathfinding algorithms can be quite expensive on the CPU, especially given the scale of the map. Such, when the player wishes to view the path, the algorithm may take a few seconds. In this case, it may be suitable to implement the pathfinding using Unity's Coroutines. The use of multiple abstract data types – where suitable – should allow me to improve the speed of the algorithm. This will allow the algorithm to run on a different thread and therefore not halt the gameplay. The A* itself is an optimisation, thus it is difficult to consider optimisations until the algorithm has been implemented, although I could possibly extend the algorithm to Theta* to show more realistic paths.

One optimisation that I have chosen to make is the implementation of my own priority queue methods. As the queue will be sorted before any new values are added, I can use a binary insertion

algorithm, being time scale $O(\log_2 n)$ for my enqueue method. The dequeue method, however, will be standard, as no optimisations can be made. I will also need to implement a method that will update the priority of existing nodes if they have been reduced. Once again, I will solve this using a divide and conquer approach for optimum performance.

The pseudocode below shows the Enqueue method, contained within the priority queue class. The class will contain one variable being the queue itself, stored as a list. The method takes a Coordinate as a parameter, being the coordinate to be inserted into the queue. The priority of each node in the queue is given by the sum of its heuristic and cost. The method then uses a binary search to insert the coordinate into the queue.

```
SUBROUTINE Enqueue(Coordinate insert)

    IF queue.Count = 0 THEN
        queue.Add(insert)
        RETURN
    ENDIF

    midpoint <- 0
    insertPriority <- insert.cost + insert.heuristic
    lowerBound <- 0
    upperBound <- queue.Count – 1

    WHILE upperBound > lowerBound
        midpoint <- (upperBound + lowerBound) / 2
        IF insertPriority > queue[midpoint].cost + queue[midpoint].heuristic THEN
            lowerBound <- midpoint + 1
        ELSEIF insertPriority < queue[midpoint].cost + queue[midpoint].heuristic THEN
            upperBound = midpoint – 1
        ELSE
            queue.Insert(midpoint + 1, insert)
            RETURN
        ENDIF
    ENDWHILE

    IF insertPriority > queue[lowerBound].cost + queue[lowerBound].heuristic THEN
        lowerBound += 1
    ENDIF

    queue.Insert(lowerBound, insert)
ENDSUB
```

Below is the UpdatePriority method, once again to be contained within the class. Like the Enqueue method it uses a binary search to find a point in the queue with priority similar or equal to the priority of the subject Coordinate. The CheckSide function is another in the class, that checks either side of the point previously found, returning the index of the subject coordinate, given that it is found on that side. Having found the subject's equivalent coordinate in the list, the equivalent is removed, and the subject is added to the list using the Enqueue method.

```
SUBROUTINE UpdatePriority(Coordinate subject, float cost)
        midpoint <- 0
        subjectPriority <- subject.cost + subject.heuristic
        lowerBound <- 0
        upperBound <- queue.Count – 1
        WHILE upperBound > lowerBound
                midpoint <- (upperBound + lowerBound) / 2
                midpointPriority <- queue[midpoint].cost + queue[midpoint].heuristic
                IF midpointPriority = subjectPriority
                        target = CheckSide(subject, midpoint, -1, subjectPriority
                        IF target = -1 THEN
                                target <- CheckSide(subject, midpoint, 1, subjectPriority)
                        ENDIF
                        queue.RemoveAt(target)
                        subject.cost <- cost
                        Enqueue(subject)
                        RETURN
                ELSEIF subjectPriority > midPointPriority THEN
                        lowerBound <- midpoint + 1
                ELSEIF subjectPriority < midPointPriority THEN
                        upperBound <- midpoint + 1
                ENDIF
        ENDWHILE

        queue.RemoveAt(lowerBound)
        subject.cost <- cost
        EnQueue(subject)
ENDSUB
```

Our final stage before we can detail the pathfinding algorithm is to consider how the position of enemies will affect the path. To do this we can use a simple radial influence map. This means that nodes within the field of view of any enemy can be given an additional cost, where nodes closer to the enemy have a greater cost than those towards the edge of the field of view. This means that our path will try to avoid these nodes and thus avoid enemies where possible. Below is a method that will determine a

preliminary cost for each node based on the current positions of all enemies, this will occur before the main A* algorithm occurs and must be called each time a path is to be found, as the position of the enemies will be constantly changing. 'graph' is a 2D array that is the initial storage location of all of the coordinates that make up the map.

```
SUBROUTINE ApplyEnemyProximity()

        FOREACH enemy IN enemies
                FOR y <- - enemy.FOVDistance TO enemy.FOVDistance
                        FOR y <- - enemy.FOVDistance TO enemy.FOVDistance
                                displacement <- Mathf.Sqrt( x * x + y * y )
                                IF displacement = 0 THEN
                                        displacement <- 1
                                ENDIF
                                graph[x + enemy.position.x + heightmap.GetLengthX / 2, y +
enemy.position.y + heightmap.GetLengthY/ 2].enemyCost <- enemy.FOVDistance / displacement *
enemyCostCoefficient + enemyCostCoefficient
                        ENDFOR
                ENDFOR
        ENDFOR

ENDSUB
```

With our cost functions and priority queue established, we can consider how the pathfinding should work. The below algorithm shows the basis of the A* algorithm, making use of the cost and priority queue we previously defined. The parameter 'target' defines a 2D position in world space on the x,z axis. As the terrain will be centred on 0,0, target can hold both as negative values, therefore, to find the index of the goal in the graph, the target's positive equivalent must be found. In the A* algorithm, the frontier is a queue of nodes to be traversed, and thus makes use of the priority queue. 'used', however, is a standard list, storing all nodes that have already been visited. The algorithm loops while the frontier isn't empty, in each iteration all new neighbouring nodes are added to the frontier, and all previously seen before have their priority updated if necessary. Once the goal is reached, the loop breaks and the path can be unwound. This is demonstrated below by the A* subroutine.

```
SUBROUTINE A*(Vector2 playerPos, Vector2 target)

        goal <- graph[target.x + heightmap.GetLengthX / 2, target.y + heightmap.GetLengthY / 2]
        current <- graph[playerPos.x + heightmap.GetLengthX / 2, playerPos.y +
heightmap.GetLengthY / 2
        frontier.EnQueue(current)
```

```
        used.Add(current)
        current.onFrontier <- TRUE
        current.cost <- 0

        WHILE frontier.Count() > 0
                current <- frontier.DeQueue()
                IF current = goal THEN
                        BREAK
                ENDIF
                current.onFrontier <- FALSE
                current.closed <- TRUE
                FOR y <- Max(0, current.y – 1) TO Min(heightmap.GetLengthY, current.y + 1)
                        FOR x <- Max(0, current.x – 1) TO Min(heightmap.GetLengthX, current.x +
1)
                                temp <- graph[x,y]
                                newcost <- 0
                                IF current = temp
                                        CONTINUE
                                ENDIF
                                IF temp.onFrontier = FALSE AND temp.closed = FALSE THEN
                                        temp.cost <- current.cost + MoveCost(current, temp)
                                        temp.heuristic = HeuristicCost(temp, goal)
                                        frontier.EnQueue(temp)
                                        used.Add(temp)
                                        temp.onFrontier <- TRUE
                                        temp.cameFrom <- current
                                ELSEIF temp.onFrontier AND temp.cost > (newcost <- current.cost
+ MoveCost(current, temp)) THEN

                                        frontier.UpdatePriority(temp, newcost)
                                        temp.cameFrom <- current
                                ENDIF
                        ENDFOR
                ENDFOR
        ENDWHILE

        WHILE current.cameFrom != NULL
                Path.Add(current.)
                Current = current.cameFrom
        ENDWHILE
        path.Reverse()
        FOR c <- 0 TO used.Count
                c.Reset()
```

```
        ENDFOR

        RETURN path
ENDSUB
```

ABSTRACT DATA

2D Array

In my program, the noise values used to create the terrain mesh are stored in a 2D array that represents a coordinate map. Such, all variables where an area of noise is stored will use this data type, as it will allow for ease of their interaction, especially during iteration. Furthermore, this ADT allows me to store all the noise map values in one place so that I can easily create vertices in 3D space and keep the code concise. Another major use of a 2D array in my program is to represent a graph during pathfinding, where each position in the 2D array is a node in the graph.

List

A list will be used as the return data type from my pathfinding function, where each item in the list is a node on the desired path. Similarly, when spawning objects, the Poisson Disc Sampling method shall return a list of coordinates to spawn objects at. The benefits of a list in these cases, is that the size of the list is unknown, thus the dynamic nature becomes very helpful.

Dictionary

I will be using dictionaries in my project, this is because it is very helpful to be able to look-up another value based on a key. For example; I will use a dictionary to look-up terrain sections via their coordinate, when checking whether their collider or rendering needs to be updated. The benefit of the dictionary is the fast look-up speed provided from the hash-table. The other use of a dictionary will be to store the player's collectable items. This is because they all share similar attributes and inherit from the class Item, allowing for quick-lookup when the player collides with something.

Queue

In my project, I will be implementing my own priority queue to assist in pathfinding. As explained earlier, using a priority queue means that nodes closer to the goal can be checked first, improving the execution time of the pathfinding algorithm. I will be storing the values in a list and use a binary insertion method to add new nodes or update previous ones.

Coordinate

This is a structure that I plan to create when implementing pathfinding into the system. I will use coordinates as nodes in the graph, and such within each struct I will store information relevant to only that coordinate. Each coordinate will store its position in world space as two integers, along with storing the height of the terrain at its position as a floating-point number. In terms of pathfinding, each coordinate will store Boolean values to detail whether the node is on the frontier, or whether the node

has been visited before. In addition, each coordinate will store a cost required to reach it from the start node, as well as a heuristic value to be used in the A* algorithm. I may also use the structure in other parts of the system if I feel that its functionality would be appropriate.

TerrainType

I will also be creating a struct called 'TerrainType'. When creating the texture for the mesh, the program needs to know what type of terrain the texture will be applied to. 'TerrainType' will be used to store data on what defines a certain type of terrain. For example, the height at which at rocks would start to appear, along with a suitable texture colour tint.

PoissonObject

This struct will be used to store data about objects that should be spawned using the PDS algorithm. Such their spawn radius and appropriate game object will be contained within this struct. Making this serializable will enable me to experiment easily using the Unity inspector.

UNITY

Unity is a game engine with a great amount of support for 3D graphics. This should be beneficial, as I won't have to acquire too much knowledge on graphics and rendering if it isn't relevant to my code. Unity also provides a fully-fledged physics engine which I will be able to make use of in my game when the player interacts with all rigid bodied objects in the world. Furthermore, Unity allows me to use my preferred language - C# - for scripting, meaning that hopefully my development time won't be seriously deterred by a learning curve.

In Unity, all scripts by default inherit from the MonoBehaviour class. This provides each script with the functionality of Unity's standard methods. Awake() is a method which is called only once whenever a script instance is loaded and is used to initialise variables and references. Start() is a method which is called immediately following Awake() or when a script becomes enabled and similarly only occurs once in the lifetime of the script. Update() and LateUpdate() are called at the beginning and end of frames respectively, and are used to implement changes that could occur in any frame. The final MonoBehaviour method that I plan on using is OnGUI() which is used to handle all GUI interface, such as clicking a button.

Scenes are used in Unity, to separate different levels or vastly different game-states. Each scene contains a collection of GameObjects, where all objects must be a GameObject. GameObjects are essentially containers and are defined by the components that they are assigned. Unity has lots of pre-built component types that when grouped form characteristic objects to make development faster, but it is possible to add almost any component to any GameObject. Each script can be added to a GameObject as a component, defining the behaviour of that GameObject, or the tasks that it should perform. This system can make OOP in Unity difficult to adhere to in cases, however in my project I will try to follow the principles wherever possible.

One of the most useful features in Unity is Prefabs. A prefab is essentially a template or basic version of a GameObject, saved as an asset in the project. It is then possible to create a clone of that Prefab, using a reference to that Prefab. This will allow me, for example, to spawn multiple trees based on a basic tree prefab. Furthermore, changes made to a Prefab can be saved to then be applied to all instance

of the prefab. This is very helpful when experimenting in the scene with different coefficients and variables.

The inspector allows you to view all the components and public fields of each GameObject in the scene. The inspector offers easy adjustment of variables and component values which can make experimenting with different values much easier, especially as Unity allows for the alteration of fields during runtime, meaning that execution doesn't have to be restarted each time the creator wants to see the effect of a change. The inspector labels variables by their name defined in the code, where a space is introduced wherever a capital letter appears in the variable name. Variables in Unity default to public for functionality in the inspector as only public fields are visible. Such, I will need to be careful when declaring variables to make sure that they are only public when necessary.

Serialisation in Unity works differently to most programs due to serialisation being possible in a real-time environment. Such, for a field to be serializable it must be public and not static, constant or read-only. It is important to understand serialisation in Unity because when a custom class is serialised, it is serialised by value in the same way as a struct. This means that if multiple references are made to the same object, those references will become separate objects during serialisation. The requirement for fields to be public to be viewed in the inspector causes a clash with OOP, however there is in cases a solution offered by Unity, this being scriptable attributes

Unity contains support for lots of different scripting attributes. These attributes differ from the standard definition of attribute in OOP. In Unity, a scriptable attribute is a marker placed above a field, class or property to indicate that it has a special behaviour. For example, to fix our problem proposed above, private fields marked with the 'SerializeField' attribute become serializable, allowing them to be altered in the inspector, given that they aren't static, constant or read-only. Similarly, the attribute 'Serializable' can be placed above a class to make it serializable. The reverse can also be achieved using the 'HideInInspector' attribute that will prevent a public field from being visible in the inspector. Further attributes include tags such as 'Range(,)' which creates a slider in the inspector that only allows the field to hold a value between the two specified in the attribute.

Coroutines are Unity's own solution for multi-threading. I will be using a co-routine to dictate the searching and shooting of enemies in the game. By using a co-routine, I can dictate events based on time intervals, rather than by the speed of execution of the rest of the program. Should I become more familiar with coroutines, I may try to optimise the terrain generation by executing it on multiple threads using the coroutine system, as the Unity API isn't thread-safe with the use of standard multi-threading.

The final major reason why I chose Unity, is due to its UI. It is very easy to create an impressive looking UI in Unity as well as integrate functionality with the program into it. My end-user Ben. E stated that the appearance of the game would play a significant factor in his impressions of it, and so I feel that Unity offers the possibility to meet those expectations. In Unity, all UI elements must be children of a Canvas GameObject. The Canvas is automatically created when the first UI element is added to scene as an empty container object with a Canvas component. The RenderMode field of the canvas component means that UI elements can be rendered in scene view, this means that their position or rotation on the screen doesn't change based on the position of the main camera. This will be very useful for implementing elements such as a pause menu or player HUD. Furthermore, the RenderMode will ensure that the Canvas automatically changes size should the screen size or resolution change, making it very easy to develop for multiple different devices. However, the adaptability of the

RenderMode makes it possible for a Canvas to be rendered like any other GameObject in the scene in world space. This can be used to create UI's that are part of the game-world. Another useful field in the Canvas component is SortOrder, this will allow be to detail the order in which UI objects should be rendered onto the screen, or rather their position on top of each other. To create a good-looking UI, it is likely that I will need to display text within a box and so I will need to consider the render order of my UI objects, especially as parents are - by default - rendered behind a child object due to being more superior in the hierarchy.

UI DESIGN

As I plan to implement my project using Unity, there will be a need for an effective UI in the system. Furthermore, I have defined the development of a user HUD in my system objectives. Thus, I have decided that creating a design for the UI will give me a clearer plan during implementation. The UI will be crucial to the player's navigation of the map, which will in turn determine their success in the game. I will display a mini-map in the upper-right corner of the screen, this will show the player's position relative to a small area around them. By finding the different collectable items in the game, the player's mini-map will be upgraded with a compass and an enemy tracker. Also displayed during gameplay will be the player's health-bar. This will be displayed in the top-right of the screen. The player loses health when hit by an enemy and can replenish it by finding any collectable item in the game. The camera in the main game-scene is not statically positioned as the player can rotate it around themselves at any time. The camera should however remain a defined distance away from the player whenever possible, as the camera may need to manoeuvre round trees. The camera should also not rotate above or below the player past a defined angle. These settings can be adjusted in the main menu, or settings pause menu. Below is an example for the UI in the main game scene while the game isn't paused.

Game Scene:

This is the player's health bar, it always shows the player's current health as a number out of 100. The fill percentage is determined by the fraction of the player's max possible health that the player currently has. The font used for the text is FredokaOne at size 14 with an RGB value of FFFFFF. The text should be anchored to the centre of the bar. The colour of the bar could be made to change based on the fraction of health remaining.

This is the player's in-game minimap. It displays a small circular area of the world surrounding the player. As shown, the player's minimap marker should be rotated to align with the direction that the camera is facing. The minimap should rotate as the camera rotates around the y(vertical)-axis. The minimap can be viewed in full in the information menu, accessed by pressing the P

The camera should always remain a set distance away from the player, by default around 5m. However, the option to alter the game settings may be added as part of an extension objective. The camera should however move to avoid obstacles, such that a clear view of the player is always maintained. The camera should render all objects and lights in the game. The trees and other game objects should cast shadows to improve the aesthetic of the game.

Inventory Menu:

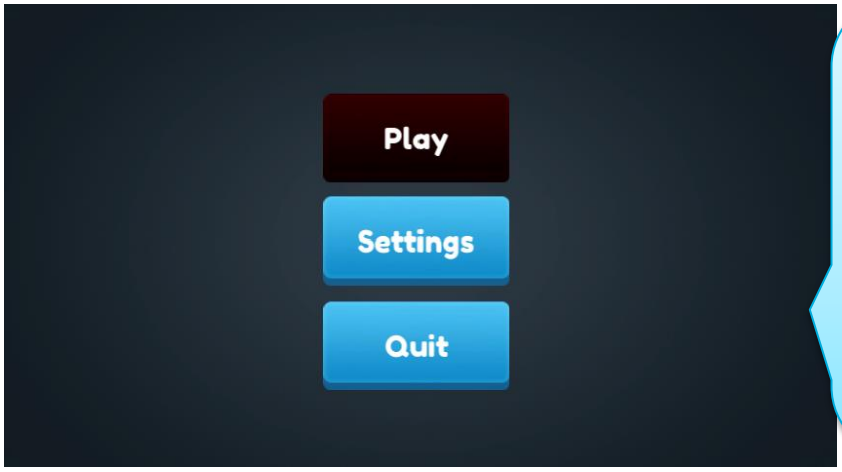
Below is an example for the UI in the inventory menu. The game will stop when the game is paused, meaning that the player won't be attacked by enemies while accessing this feature. To show that the game is in the inventory menu a dark-filter should be applied over the scene to distinguish it from normal gameplay. While accessing the inventory menu, the player's health bar should remain visible. Such, if I add consumable items, this will allow the player to quickly check their health before using any. The global minimap displayed in the inventory menu should differ from the in-game minimap in

the sense that certain objects such as trees aren't rendered. I feel this will improve the appearance of the minimap as well as considerably improve performance on large maps.

Above is a space dedicated to displaying the player's inventory and allowing them to manage it. It is unlikely that I will come to implement this, as introducing a combat system would be very demanding and require animation skill that I don't currently have. Despite this I have designed the inventory should I come to it. The inventory space should be 400 and 300 UI units wide and high respectively. This will allow for 8 columns of spaced inventory slots across 6 rows. With each slot being 30 UI units in each dimension.

This is an example of the player's large scale minimap. This should be approximately 360 UI units in both dimensions, this should allow it to fit comfortable on screen alongside the inventory. This minimap should display the entirety of the world and when active should disable the in-game minimap. When the player acquires collectables such as the Enemy Finder, this minimap should detail the positions of all the enemies in the map. Furthermore, once the player has acquired the Pathfinder, they will be able to use this minimap to view a suggested path to the goal. With another collectable, the player will also be able to see their positions in relation to the whole map, shown by a marker.

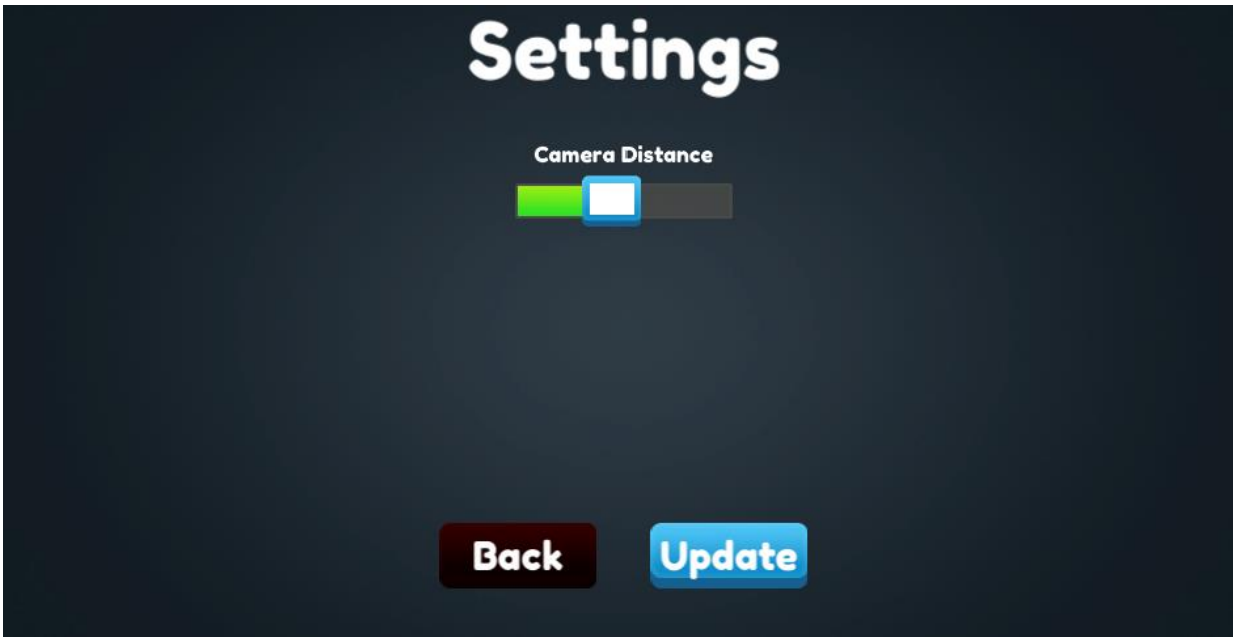
Main Menu:



This is an example of the main-menu scene that will appear when the game begins. I plan to keep the UI in the menus simple, so I can spend more time focusing on functionality within the game.

The buttons should be coloured when highlighted or pressed to alert the user of which option they are about to select. Pressing the play button should take the user to the main game-scene, whilst pressing the quit button should end the application immediately. The settings button will take the user to a settings menu, that will most likely be developed during extension tasks. Each of the buttons should use the font FredokaOne in size 30, with an RGB value of FFFFFF. This should keep a theme within the game which will hopefully improve its aesthetic and make the system feel unified.

Settings Menu:



I have designed the settings menu as predominantly empty space. This is because I don't plan to implement many settings unless I come to the extension tasks. I have however added a slider that will allow the user to adjust the standard distance of the camera from the player. The user must also click the Update button if they would like the changes to be saved. I shall include this settings menu in the

same scene as the main-menu to make transitions between them slightly easier. Once again, the font used is FredokaOne with an RGB value of FFFFFF. For the buttons I plan to use size 30, and size 60 for the title. The 'Back' button will be pre-selected and clicking it will return the user to the main-menu again. To implement changes made in the settings menu, I plan to make use of Unity's PlayerPrefs feature. This will automatically create a file storing all of the PlayerPrefs that I have defined. When the game is next run, all of the player's settings can then be loaded from that file. I will not know, until implementation, whether this is more suitable than a database.

Solution Development

ANNOTATED CODE LISTING

Map Generation

UpdateableData.cs

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

public class UpdateableData : ScriptableObject {

	public event System.Action OnValuesUpdated;
	public bool autoUpdate;

	#if UNITY_EDITOR

	public void NotifyOfUpdatedValues()
	{
		if (OnValuesUpdated != null)
		{
			UnityEditor.EditorApplication.update -= NotifyOfUpdatedValues;
			OnValuesUpdated();
		}
	}

	protected virtual void OnValidate()
	{
		if(autoUpdate)
		{
			UnityEditor.EditorApplication.update += NotifyOfUpdatedValues;
		}
	}
	#endif

}
```

NoiseData.cs

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

[CreateAssetMenu()]
public class NoiseData : UpdateableData {

	// data storage class
	[SerializeField]
	private float noiseScale;
	public float getNoiseScale()
	{
		return noiseScale;
	}
```

```
        [SerializeField]
        private int octaves;
        public int getOctaves()
        {
                return octaves;
        }

        [Range(0, 1), SerializeField]
        private float persistence;
        public float getPersistence()
        {
                return persistence;
        }

        [SerializeField]
        private float lacunarity;
        public float getLacunarity()
        {
                return lacunarity;
        }

        [SerializeField]
        private int seed;
        public int getSeed()
        {
                return seed;
        }

        [SerializeField]
        private Vector2 offset;
        public Vector2 getOffset()
        {
                return offset;
        }
        public float getOffsetX()
        {
                return offset.x;
        }
        public float getOffsetY()
        {
                return offset.y;
        }

        [SerializeField]
        private bool useFalloff;
        public bool UseFalloff()
        {
                return useFalloff;
        }

        [SerializeField]
        private MapGen.FalloffFunction falloffFunction;
        public MapGen.FalloffFunction getFalloffFunction()
        {
                return falloffFunction;
        }

#if UNITY_EDITOR
```

```
	protected override void OnValidate()
	{
		if (lacunarity < 1)
		{
			lacunarity = 1;
		}
		if (octaves < 0)
		{
			octaves = 1;
		}

		base.OnValidate();
	}

#endif

}
```

TerrainData.cs

```
using System.Collections;
using UnityEngine;

[CreateAssetMenu()]
public class TerrainData : UpdateableData {

	//data storage class
	[SerializeField]
	private float meshHeightMultiplier;
	public float getMeshHeightMultiplier()
	{
		return meshHeightMultiplier;
	}

	[SerializeField]
	private AnimationCurve meshHeightCurve;
	public AnimationCurve getMeshHeightCurve()
	{
		return meshHeightCurve;
	}

	[SerializeField]
	private bool useFlatShading;
	public bool UseFlatShading()
	{
		return useFlatShading;
	}

	[SerializeField]
	private float uniformScale = 1f;
	public float getUniformScale()
	{
		return uniformScale;
	}

	[SerializeField]
```

```
        private Vector2 mapDimension;
        public Vector2 getMapDimension()
        {
                return mapDimension;
        }

        public float minHeight
        {
                get
                {
                        return uniformScale * meshHeightMultiplier *
meshHeightCurve.Evaluate(0);
                }
        }
        public float maxHeight
        {
                get
                {
                        return uniformScale * meshHeightMultiplier *
meshHeightCurve.Evaluate(1);
                }
        }

}
```

TextureData.cs

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

[CreateAssetMenu()]
public class TextureData : UpdateableData {

        //mainly data storage class
        const int textureSize = 512;
        const TextureFormat textureFormat = TextureFormat.RGB565;

        public Layer[] layers;

        float savedMinHeight;
        float savedMaxHeight;

        public void ApplyToMaterial(Material material)
        {
                material.SetInt("layerCount", layers.Length);
                material.SetColorArray("baseColours", layers.Select(x =>
x.tint).ToArray());
                material.SetFloatArray("baseStartHeights", layers.Select(x =>
x.startHeight).ToArray());
                material.SetFloatArray("baseBlends", layers.Select(x =>
x.blendStrength).ToArray());
                material.SetFloatArray("baseColourStrength", layers.Select(x =>
x.tintStrength).ToArray());
                material.SetFloatArray("baseTextureScales", layers.Select(x =>
x.textureScale).ToArray());
```

```
            Texture2DArray texturesArray = GenerateTextureArray(layers.Select(x =>
x.texture).ToArray());
            material.SetTexture("baseTextures", texturesArray);


            UpdateMeshHeights(material, savedMinHeight, savedMaxHeight);
        }

        public void UpdateMeshHeights(Material material, float minHeight, float
maxHeight)
        {
            savedMinHeight = minHeight;
            savedMaxHeight = maxHeight;

            material.SetFloat("minHeight", minHeight);
            material.SetFloat("maxHeight", maxHeight);
        }

        Texture2DArray GenerateTextureArray(Texture2D[] textures)
        {
            Texture2DArray textureArray = new Texture2DArray(textureSize,
textureSize, textures.Length, textureFormat, true);
            for (int i = 0; i < textures.Length; i++)
            {
                textureArray.SetPixels(textures[i].GetPixels(), i);
            }
            textureArray.Apply();
            return textureArray;
        }

        [System.Serializable]
        public class Layer
        {
            public Texture2D texture;
            public Color tint;
            [Range(0,1)]
            public float tintStrength;
            [Range(0, 1)]
            public float startHeight;
            [Range(0, 1)]
            public float blendStrength;
            public float textureScale;
        }
}
```

NoiseGen.cs

```
using UnityEngine;
using System.Collections;

public static class NoiseGen
{
    static float maxNoiseHeight = float.MinValue;
    static float minNoiseHeight = float.MaxValue;
    public static float[,] GenerateNoiseMap(int mapWidth, int mapHeight, int seed,
float scale, int octaves, float persistance, float lacunarity, Vector2 offset)
    {
        float[,] noiseMap = new float[mapWidth, mapHeight];
```

```
        System.Random prng = new System.Random(seed);
        Vector2[] octaveOffsets = new Vector2[octaves];
        for (int i = 0; i < octaves; i++)
        {
            float offsetX = prng.Next(-100000, 100000) + offset.x;
            float offsetY = prng.Next(-100000, 100000) - offset.y;
            octaveOffsets[i] = new Vector2(offsetX, offsetY);
        }

        if (scale <= 0)
        {
                    // scale must be positive value
            scale = 0.0001f;
        }

        for (int y = 0; y < mapHeight; y++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                        // initialising noise constants
                float amplitude = 1;
                float frequency = 1;
                float noiseHeight = 0;
                for (int i = 0; i < octaves; i++)
                {
                                // Finds noise coordinate to be sampled
                                // Difference between consecutive samples
determined by frequency and noise scale
                    float sampleX = (x - (mapWidth/2f) + octaveOffsets[i].x) / scale *
frequency;
                    float sampleY = (y - (mapHeight/2f) + octaveOffsets[i].y) / scale
* frequency;

                                // takes noise value at sample coordinate
                    float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;
                                // adjusts noise value by current amplitude
                    noiseHeight += perlinValue * amplitude;

                                // applies persistance and lacunarity between
octaves
                    amplitude *= persistance;
                    frequency *= lacunarity;
                }
                        // finds overall max and min heights
                if (noiseHeight > maxNoiseHeight)
                {
                    maxNoiseHeight = noiseHeight;
                }
                else if (noiseHeight < minNoiseHeight)
                {
                    minNoiseHeight = noiseHeight;
                }
                        // stores noise value at relevant position in array
                noiseMap[x, y] = noiseHeight;
            }
        }
```

```
        for (int y = 0; y < mapHeight; y++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                            // standarises noise heights between min and max heights
                noiseMap[x, y] = Mathf.InverseLerp(minNoiseHeight, maxNoiseHeight,
noiseMap[x, y]);
            }
        }

        return noiseMap;
    }


}
```

FalloffGen.cs

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

public static class FalloffGen{


    public static float[,] GenerateFalloffMap(int size, Vector2 mapDimension,
MapGen.FalloffFunction falloffFunction)
    {

              float[,] map = new float[size * (int)mapDimension.x + 2, size *
(int)mapDimension.y + 2];
              // iterations included noise borders
              for (int i = 0; i < size* mapDimension.x + 2; i++)
        {
            for (int j = 0; j < size * mapDimension.y + 2 ; j++)
            {
                float x = j / (float)(size * mapDimension.y) * 2 - 1;
                float y = i / (float)(size * mapDimension.x) * 2 - 1;

                float value = Mathf.Max(Mathf.Abs(x), Mathf.Abs(y));

                            // chooses falloff equation to used
                            if (falloffFunction == MapGen.FalloffFunction.Sigmoid)
                            {
                                  map[i, j] = EvaluateSigmoid(value, mapDimension);
                            }
                            else if(falloffFunction ==
MapGen.FalloffFunction.Standard)
                            {
                                  map[i, j] = EvaluateStandard(value, mapDimension);
                            }
            }
        }
        return map;
    }
    public static float EvaluateStandard(float value, Vector2 mapDimension)
```

```
    {
        float a = 3f;
        float b = (float)mapDimension.x;

        return Mathf.Pow(value, a) / (Mathf.Pow(value, a) + Mathf.Pow(b - b * value,
a));
    }
    public static float EvaluateSigmoid(float value, Vector2 mapDimension)
    {
        float a = -4 * (float)mapDimension.x;
        return  Mathf.Exp(6 * value - 1) / (Mathf.Exp(6 * value - 1) +
Mathf.Pow(value,a));
    }

}
```

MeshGen.cs

```
using UnityEngine;
using System.Collections;

public class MeshData
{
       private Vector3[] vertices;
       int[] triangles;
       Vector2[] uvs;

       Vector3[] borderVertices;
       int[] borderTriangles;

       int triangleIndex;
       int borderTriangleIndex;
       bool useFlatShading;

       public MeshData(int verticesPerLine, float[,] heightMap, float
heightMultiplier, AnimationCurve _heightCurve, bool useFlatShading)
       {

              AnimationCurve heightCurve = new AnimationCurve(_heightCurve.keys);
              int borderedSize = heightMap.GetLength(0);
              int meshSize = borderedSize - 2;
              float topLeftX = (meshSize - 1) / -2f;
              float topLeftZ = (meshSize - 1) / 2f; // mesh generation begins from
top-left of array

              this.useFlatShading = useFlatShading;
              vertices = new Vector3[verticesPerLine * verticesPerLine];
              uvs = new Vector2[verticesPerLine * verticesPerLine];
              triangles = new int[(verticesPerLine - 1) * (verticesPerLine - 1) * 6];

              borderVertices = new Vector3[verticesPerLine * 4 + 4];
              borderTriangles = new int[24 * verticesPerLine];

              int[,] vertexIndicesMap = new int[borderedSize, borderedSize];
              // Stores the number of vertices that make up the mesh
              int meshVertexIndex = 0;
              // Stores the number of vertices found in the border, negative increment
```

```
		int borderVertexIndex = -1;

		// Updates index depending whether the vertex makes up the border or not
		for (int y = 0; y < borderedSize; y++)
		{
			for (int x = 0; x < borderedSize; x++)
			{
				bool isBorderVertex = y == 0 || y == borderedSize - 1 || x
== 0 || x == borderedSize - 1;

				if (isBorderVertex)
				{
					vertexIndicesMap[x, y] = borderVertexIndex;
					borderVertexIndex--;
				}
				else
				{
					vertexIndicesMap[x, y] = meshVertexIndex;
					meshVertexIndex++;
				}
			}
		}

		for (int y = 0; y < borderedSize; y++)
		{
			for (int x = 0; x < borderedSize; x++)
			{
				int vertexIndex = vertexIndicesMap[x, y];
				Vector2 percent = new Vector2((x - 1) / (float)meshSize,
(y - 1) / (float)meshSize);

				// finds vertex position in game space
				float vertexHeight = heightCurve.Evaluate(heightMap[x, y])
* heightMultiplier;
				Vector3 vertexPosition = new Vector3(topLeftX + percent.x
* meshSize, vertexHeight, topLeftZ - percent.y * meshSize);

				AddVertex(vertexPosition, percent, vertexIndex);

				if (x < borderedSize - 1 && y < borderedSize - 1)
				{
					// creates the 2 triangles based on vertex
					// each set of four vertices creates two triangles
					int a = vertexIndicesMap[x, y];
					int b = vertexIndicesMap[x + 1, y];
					int c = vertexIndicesMap[x, y + 1];
					int d = vertexIndicesMap[x + 1, y + 1];
					AddTriangle(a, d, c);
					AddTriangle(d, a, b);

				}
				vertexIndex++;
			}
		}

		ProcessMesh();

	}
```

```
        //Constructor chaining
        public MeshData(float[,] heightMap, float heightMultiplier, AnimationCurve
_heightCurve, bool useFlatShading)
                : this(heightMap.GetLength(0), heightMap, heightMultiplier,
_heightCurve, useFlatShading) { }


        public void AddVertex(Vector3 vertexPosition, Vector2 uv, int vertexIndex)
        {
                if(vertexIndex < 0)
                {
                        // vertex is found in border and added to border array
                        borderVertices[-vertexIndex - 1] = vertexPosition;
                }
                else
                {
                        // vertex is found in mesh and added to vertices array
                        vertices[vertexIndex] = vertexPosition;
                        uvs[vertexIndex] = uv;
                }
        }

        public void AddTriangle(int a, int b, int c)
        {
                // a,b and c represent an index in the vertices array

                if (a < 0 || b < 0 || c < 0)
                {
                        borderTriangles[borderTriangleIndex] = a;
                        borderTriangles[borderTriangleIndex + 1] = b;
                        borderTriangles[borderTriangleIndex + 2] = c;
                        borderTriangleIndex += 3;
                }
                else
                {
                        triangles[triangleIndex] = a;
                        triangles[triangleIndex + 1] = b;
                        triangles[triangleIndex + 2] = c;
                        triangleIndex += 3;
                }
        }
        Vector3[] CalculateNormals()
        {
                // manual normal calculation to account for border
                Vector3[] vertexNormals = new Vector3[vertices.Length];
                int triangleCount = triangles.Length / 3;
                for (int i = 0; i < triangleCount; i++)
                {
                        // finds normals for all triangles in mesh
                        int normalTriangleIndex = i * 3;
                        int vertexIndexA = triangles[normalTriangleIndex];
                        int vertexIndexB = triangles[normalTriangleIndex + 1];
                        int vertexIndexC = triangles[normalTriangleIndex + 2];

                        // each triangle has a normal based on the average normal of its
vertices
                        // each vertex has a normal based on all of the triangles that it
is found in
```

```
                        Vector3 triangleNormal = SurfaceNormalFromIndices(vertexIndexA,
vertexIndexB, vertexIndexC);
                        vertexNormals[vertexIndexA] += triangleNormal;
                        vertexNormals[vertexIndexB] += triangleNormal;
                        vertexNormals[vertexIndexC] += triangleNormal;
                }

                int borderTriangleCount = borderTriangles.Length / 3;
                for (int i = 0; i < borderTriangleCount; i++)
                {
                        //finds normals for all triangles that have a vertex in the
border
                        int normalTriangleIndex = i * 3;
                        int vertexIndexA = borderTriangles[normalTriangleIndex];
                        int vertexIndexB = borderTriangles[normalTriangleIndex + 1];
                        int vertexIndexC = borderTriangles[normalTriangleIndex + 2];

                        // checks which vertices are found in the border, applys normals
to all those that aren't
                        Vector3 triangleNormal = SurfaceNormalFromIndices(vertexIndexA,
vertexIndexB, vertexIndexC);
                        if(vertexIndexA > 0)
                        {
                                vertexNormals[vertexIndexA] += triangleNormal;
                        }
                        if(vertexIndexB > 0)
                        {
                                vertexNormals[vertexIndexB] += triangleNormal;
                        }
                        if(vertexIndexC > 0)
                        {
                                vertexNormals[vertexIndexC] += triangleNormal;
                        }
                }

                for (int i = 0; i < vertexNormals.Length; i++)
                {
                        // normalises normals as sum may exceed 1
                        vertexNormals[i].Normalize();
                }
                return vertexNormals;
        }

        void FlatShading()
        {
                Vector3[] flatShadedVertices = new Vector3[triangles.Length];
                Vector2[] flatShadedUvs = new Vector2[triangles.Length];

                for (int i = 0; i < triangles.Length; i++)
                {
                        flatShadedVertices[i] = vertices[triangles[i]];
                        flatShadedUvs[i] = uvs[triangles[i]];
                        triangles[i] = i;
                }
                vertices = flatShadedVertices;
                uvs = flatShadedUvs;
        }
```

```
        public void ProcessMesh()
        {
                // normal calculation depends on lighting method
                if(useFlatShading)
                {
                        FlatShading();
                }
                else
                {
                        CalculateNormals();
                }
        }

        Vector3 SurfaceNormalFromIndices(int indexA, int indexB, int indexC)
        {
                // finds triangle normal from the vertices that make it up
                Vector3 vertexA = (indexA < 0) ? borderVertices[-indexA - 1] :
vertices[indexA];
                Vector3 vertexB = (indexB < 0) ? borderVertices[-indexB - 1] :
vertices[indexB];
                Vector3 vertexC = (indexC < 0) ? borderVertices[-indexC - 1] :
vertices[indexC];

                Vector2 sideAB = vertexB - vertexA;
                Vector3 sideAC = vertexC - vertexA;
                // uses cross-product to find perpendicular vector
                return Vector3.Cross(sideAB, sideAC).normalized;
        }

        public Mesh CreateMesh()
        {
                Mesh mesh = new Mesh();
                mesh.vertices = vertices;
                mesh.triangles = triangles;
                mesh.uv = uvs;
                if(useFlatShading)
                {
                        mesh.RecalculateNormals();
                }
                else
                {
                        mesh.normals = CalculateNormals();
                }
                return mesh;
        }


}
```

MapGen.cs

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

```
public class MapGen : MonoBehaviour
{

	public enum DrawMode { NoiseMap, Mesh };
	private DrawMode drawMode;
	public enum FalloffFunction { Standard, Sigmoid }

	public TerrainData terrainData;
	public NoiseData noiseData;
	public TextureData textureData;

	public Material terrainMaterial;


	[SerializeField]
	private int maxViewDistance;
	public int getMaxViewDistance()
	{
		return maxViewDistance;
	}

	[SerializeField]
	private float colliderDistanceThreshold;
	public float getColliderDistanceThreshold()
	{
		return colliderDistanceThreshold;
	}

	[SerializeField]
	private float treeDistanceThreshold;
	public float getTreeDistanceThreshold()
	{
		return treeDistanceThreshold;
	}


    public float[,] falloffMap, wholeNoiseMap, realMap;
	public int smallCoordinator, globalCoordinator;



	public bool autoUpdate;

	public int mapSectionSize
	{
		get
		{
			if(terrainData.UseFlatShading())
			{
				return 95;
			}
			else
			{
				return 239;
			}
		}
	}
```

```
    private void Start()
    {
            PreGenerateWholeMap();
            textureData.ApplyToMaterial(terrainMaterial);
       }

       void OnValuesUpdated()
       {
            if(!Application.isPlaying)
            {
                  DrawMapInEditor(); // used to preview map changes in  Unity
editor
            }
       }

       void OnTextureValuesUpdated ()
       {
            textureData.ApplyToMaterial(terrainMaterial);
       }

       void OnValidate() // prevents null errors from occurring when altering or
updating terrain generation variables in the inspector
       {

            if(terrainData != null)
            {
                  terrainData.OnValuesUpdated -= OnValuesUpdated;
                  terrainData.OnValuesUpdated += OnValuesUpdated;
            }
            if (noiseData != null)
            {
                  noiseData.OnValuesUpdated -= OnValuesUpdated;
                  noiseData.OnValuesUpdated += OnValuesUpdated;
            }
            if(textureData != null)
            {
                  textureData.OnValuesUpdated -= OnTextureValuesUpdated;
                  textureData.OnValuesUpdated += OnTextureValuesUpdated;
            }
            falloffMap = FalloffGen.GenerateFalloffMap(mapSectionSize,
terrainData.getMapDimension(), noiseData.getFalloffFunction());
       }

       public void PreGenerateWholeMap()
       {
            // generates the noisemap for the entire map
            smallCoordinator = -((int)terrainData.getMapDimension().x - 1) / 2;
            globalCoordinator = (mapSectionSize) * smallCoordinator;
            // generates falloff map
            falloffMap = FalloffGen.GenerateFalloffMap(mapSectionSize,
terrainData.getMapDimension(), noiseData.getFalloffFunction());
            // generates noise map
            wholeNoiseMap = NoiseGen.GenerateNoiseMap(mapSectionSize *
(int)terrainData.getMapDimension().x + 2, mapSectionSize *
(int)terrainData.getMapDimension().y + 2, noiseData.getSeed(),
noiseData.getNoiseScale(), noiseData.getOctaves(), noiseData.getPersistence(),
noiseData.getLacunarity(), noiseData.getOffset());
```

```
            realMap = new float[wholeNoiseMap.GetLength(0),
wholeNoiseMap.GetLength(0)];
            for (int y = 0; y < mapSectionSize * terrainData.getMapDimension().y +
2; y++)
            {
                  for (int x = 0; x < mapSectionSize *
terrainData.getMapDimension().x + 2; x++)
                  {
                        if (noiseData.UseFalloff())
                        {
                              // applies falloff map to noisemap
                              wholeNoiseMap[x, y] =
Mathf.Clamp01(wholeNoiseMap[x, y] - falloffMap[x, y]);
                        }
                        // adjusts heights using animation curve
                        realMap[x,y] =
terrainData.getMeshHeightCurve().Evaluate(wholeNoiseMap[x, y]);
                        // increases heights by multiplyer suitable for gameplay
                        realMap[x, y] *= terrainData.getMeshHeightMultiplier();
                  }
            }
      }


      public void DrawMapInEditor()
    {
            PreGenerateWholeMap();
       MapData mapData = GenerateMapData(Vector2.zero);

            MapDisplay display = FindObjectOfType<MapDisplay>();
       if (drawMode == DrawMode.NoiseMap)
       {
           display.DrawTexture(TextureGen.TextureFromNoiseMap(mapData.heightMap));
       }
       else if (drawMode == DrawMode.Mesh)
       {
           display.DrawMesh(new MeshData(mapData.heightMap,
terrainData.getMeshHeightMultiplier(), terrainData.getMeshHeightCurve(),
terrainData.UseFlatShading()));
       }
    }

    void Update()
    {
    }

    public MapData GenerateMapData(Vector2 centre)
    {
            // finds the section of noise relevant to section, then applies noise
            // old : float[,] noiseMap = NoiseGen.GenerateNoiseMap(mapSectionSize,
mapSectionSize, seed, noiseScale, octaves, persistance, lacunarity, centre + offset);
            float[,] noiseMap = new float[mapSectionSize + 2, mapSectionSize + 2];
            for (int y = 0; y < mapSectionSize + 2; y++)
            {
                  for (int x = 0; x < mapSectionSize + 2; x++)
                  {
                        // converts from negative to positive, allowing noisemap
to be indexed
```

```
                            int noiseFinderX = x + (int)centre.x - globalCoordinator;
                            int noiseFinderY = y - (int)centre.y - globalCoordinator;
// subtraction of centre occurs due to top down nature of mesh generation
                            noiseMap[x, y] = wholeNoiseMap[noiseFinderX,
noiseFinderY];
                        }
                }
                // applies heights to terrain texture
                textureData.UpdateMeshHeights(terrainMaterial, terrainData.minHeight,
terrainData.maxHeight);
                // returns noise-map for mesh generation
            return new MapData(noiseMap);
        }


}

[System.Serializable]
public struct TerrainType
{
       [SerializeField]
    private string name;
       [SerializeField]
    private float height;
       [SerializeField]
    private Color colour;
}

public struct MapData
{
    public readonly float[,] heightMap;

    public MapData(float[,] heightMap)
    {
        this.heightMap = heightMap;
    }

}
```

TextureGen.cs

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

public static class TextureGen {

// used in earlier development to apply plain coloured texture to terrain
    public static Texture2D TextureFromColourMap(Color[] colourMap, int width, int
height)
    {
        Texture2D texture = new Texture2D(width, height);
        texture.filterMode = FilterMode.Point;
        texture.wrapMode = TextureWrapMode.Clamp;
        texture.SetPixels(colourMap);
        texture.Apply();
```

```
        return texture;
    }

    public static Texture2D TextureFromNoiseMap(float[,] noiseMap)
    {
        int width = noiseMap.GetLength(0);
        int height = noiseMap.GetLength(1);

        Texture2D texture = new Texture2D(width, height);

        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,
noiseMap[x, y]);
            }
        }

        return TextureFromColourMap(colourMap, width, height);

    }

}
```

TerrainSections.cs

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

public class TerrainSections : MonoBehaviour
{
       const float scale = 1f;
       [SerializeField]
       private Transform viewer;
       [SerializeField]
       private Material mapMaterial;
       [SerializeField]
       private static Vector2 viewerPosition;
       static MapGen mapGen;
       static Spawner spawner;
       private Vector2 mapDimension;
       int sectionSize;

       // Allows a terrain section to be found from its coordinate
       public Dictionary<Vector2, TerrainSection> terrainSectionDictionary = new
Dictionary<Vector2, TerrainSection>();

       void Start()
    {
        mapGen = FindObjectOfType<MapGen>();
              spawner = FindObjectOfType<Spawner>();
        sectionSize = mapGen.mapSectionSize - 1;
        mapDimension = mapGen.terrainData.getMapDimension();
              CreateMapSections();
```

```
    }

    void Update()
    {
        viewerPosition = new Vector2(viewer.position.x, viewer.position.z) / scale;
        UpdateVisibleChunks();
    }

    void UpdateVisibleChunks()
    {
            // Checks each terrain section
        for (int y = -(int)((mapDimension.y - 1) / 2); y <= (int)((mapDimension.y - 1)
/ 2); y++)
        {
                    for (int x = -(int)((mapDimension.x - 1) / 2); x <=
(int)((mapDimension.x - 1) / 2); x++)
                    {

                            Vector2 viewedSectionCoordinate = new Vector2(x, y);

       terrainSectionDictionary[(viewedSectionCoordinate)].UpdateTerrainSection();
                    }
        }
    }
       public void CreateMapSections()
       {
              // creates map sections centred around the origin
              for (int y = mapGen.smallCoordinator; y <= -mapGen.smallCoordinator;
y++)
              {
                    for (int x = mapGen.smallCoordinator; x <= -
mapGen.smallCoordinator; x++)
                    {
                            Vector2 sectionCoordinate = new Vector2(x, y);
                            // Creates section and adds it to dictionary
                            terrainSectionDictionary.Add(sectionCoordinate, new
TerrainSection(sectionCoordinate, sectionSize, transform, mapMaterial));
                    }
              }
              // Once terrain is generated, the objects can be spawned
              spawner.Spawn();
       }

       public class TerrainSection
       {
              static int instanceCounter = 0;
              GameObject meshObject;
              Vector2 position;
              Bounds bounds;


              MeshRenderer meshRenderer;
              public MeshFilter meshFilter;
              public MeshCollider meshCollider;
              public MeshData meshData;
              public MapData mapData;
```

```
            public TerrainSection(Vector2 coord, int size, Transform parent,
Material material)
            {
                  // Constructor instantiates a new terrain section with
appropriate mesh

                  // Finds global position from coordinate and section size
                  position = coord * size;
                  bounds = new Bounds(position, Vector2.one * size);
                  Vector3 positionV3 = new Vector3(position.x, 0, position.y);
                  meshObject = new GameObject("Terrain Section " +
++instanceCounter)
                  {
                        // Tag can be used for object spawning when checking
colliders
                        tag = "Terrain Section"
                  };
                  // Adds required mesh components
                  meshRenderer = meshObject.AddComponent<MeshRenderer>();
                  meshFilter = meshObject.AddComponent<MeshFilter>();
                  meshCollider = meshObject.AddComponent<MeshCollider>();

                  meshRenderer.material = material;
                  meshObject.transform.position = positionV3 * scale;
                  meshObject.transform.parent = parent;
                  meshObject.transform.localScale = Vector3.one * scale;

                  // Gets respective noise values for terrain section
                  mapData = mapGen.GenerateMapData(position);

                  // Creates section mesh from noise values
                  meshData = new MeshData(mapData.heightMap,
mapGen.terrainData.getMeshHeightMultiplier(), mapGen.terrainData.getMeshHeightCurve(),
mapGen.terrainData.UseFlatShading());

                  // Applies mesh
                  meshFilter.mesh = meshData.CreateMesh();
                  meshCollider.sharedMesh = meshFilter.mesh;
            }



            public void UpdateTerrainSection()
            {
                  // Only render terrain section if it is within the view threshold
                  float viewerDstFromNearestEdge =
Mathf.Sqrt(bounds.SqrDistance(viewerPosition));
                  bool visible = viewerDstFromNearestEdge <=
mapGen.getMaxViewDistance();
                  // uses main camera's layer mask to determine rendering
                  if(!visible)
                  {
                        meshObject.layer = 15;
                  }
                  else
                  {
```

```
                        meshObject.layer = 1;
                }
        }

        public void SetVisible(bool visible)
        {
                meshObject.SetActive(visible);
        }

        public bool IsVisible()
        {
                return meshObject.activeSelf;
        }

    }

}
```

Spawner.cs

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


public class Spawner : MonoBehaviour {

    [SerializeField]
    private GameObject player;
    [SerializeField]
    private GameObject cameraBase;
    [SerializeField]
    private PoissonObject trees;
    [SerializeField]
    private PoissonObject enemies;
    [SerializeField]
    private GameObject[] plants;
    public GameObject Rock;
    MapGen mapGen;
    private float topLeftX, topLeftZ;

    RaycastHit hit;

    public static List<Vector2> points = new List<Vector2>();
    public float iniEnemies;

    // Use this for initialization
    void Start () {
    }

    // Update is called once per frame
    void Update () {

    }
```

```
	static bool IsValid(Vector2 candidate, Vector2 sampleRegionSize, float
cellSize, float r, List<Vector2> points, int[,] grid)
	{
		// Checks whether a point is within the sample area and whether it is
within the radius of any others
		if (candidate.x >= 0 && candidate.x < sampleRegionSize.x && candidate.y
>= 0 && candidate.y < sampleRegionSize.y)
		{
			// Finds the cell that the candidate is found in
			// Meaning only surrounding 24 need be checked
			int cellX = (int)(candidate.x / cellSize);
			int cellY = (int)(candidate.y / cellSize);
			// Min and Max prevent errors when candidate is found on the edge
			int startX = Mathf.Max(0, cellX - 2);
			int startY = Mathf.Max(0, cellY - 2);
			int endX = Mathf.Min(cellX + 2, grid.GetLength(0) - 1);
			int endY = Mathf.Min(cellY + 2, grid.GetLength(1) - 1);

			for (int x = startX; x <= endX; x++)
			{
				for (int y = startY; y <= endY; y++)
				{
					int pointIndex = grid[x, y] - 1;
					if (pointIndex != -1)
					{
						// -1 marks an empty cell as indexing will
begin at 0
						float sqrDst = (candidate -
points[pointIndex]).sqrMagnitude;
						if (sqrDst < r*r)
						{
							// candidate is within the radius of
an existing point
							// candidate is therefore rejected
							return false;
						}
					}
				}
			}
			// candidate placement is valid and is accepted
			return true;
		}
		// candidate wasnt in sample area therefore rejected
		return false;
	}


	public List<Vector2> GeneratePoints(float r, Vector2 sampleRegionSize, int k)
	{
		// Generates a list of points where each point isnt within the radius of
any others
		// r stores the radius of the points, k is the number of samples to be
tested before rejection

		int n = 2; // used to store number of dimensions for sampling in
		float cellSize = r / Mathf.Sqrt(n); // cellsize is defined from radius
of points
```

```
            int[,] grid = new int[Mathf.CeilToInt(sampleRegionSize.x / cellSize),
Mathf.CeilToInt(sampleRegionSize.y / cellSize)];
            List<Vector2> activeList = new List<Vector2>
            {
                  new Vector2(UnityEngine.Random.Range(1, sampleRegionSize.x - 1),
UnityEngine.Random.Range(1, sampleRegionSize.y - 1))
            };
            while (activeList.Count > 0)
            {
                  // iterates while there are still spawn points to be tested
                  int activeListIndex = UnityEngine.Random.Range(0,
activeList.Count);
                  Vector2 activeSpawnPoint = activeList[activeListIndex];
                  bool candidateAccepted = false;

                  for (int i = 0; i < k; i++)
                  {
                        // Generates a random position between r and 2r of the
spawn point
                        Vector2 candidate = activeSpawnPoint + AnnulusOffset(r,
2*r);

                        // checks whether a point at the generated position would
be valid
                        if (IsValid(candidate, sampleRegionSize, cellSize, r,
points, grid))

                        {

// if valid then the point is stored and becomes a future spawn point
                              points.Add(candidate);
                              activeList.Add(candidate);
                              // stores point's index at the cell where the point
is rounded to
                              grid[(int)(candidate.x / cellSize),
(int)(candidate.y / cellSize)] = points.Count;
                              candidateAccepted = true;
                              break;
                        }
                  }
                  if (!candidateAccepted)
                  {
                        // no points could be spawned around the active point
                        activeList.RemoveAt(activeListIndex);
                  }

            }

            return points;
      }

      public void SpawnPoisson(GameObject[] gameObjectArray, float r, int k)
      {
            mapGen = FindObjectOfType<MapGen>();
            // Spawns objects at generated points
            float height = 0f;
```

```
            AnimationCurve heightCurve = new
AnimationCurve(mapGen.terrainData.getMeshHeightCurve().keys);
            List<Vector2> points = GeneratePoints(r,
mapGen.terrainData.getMapDimension() * mapGen.mapSectionSize, k);
            topLeftX = ((mapGen.mapSectionSize - 1) *
mapGen.terrainData.getMapDimension().y) / -2f;
            topLeftZ = ((mapGen.mapSectionSize - 1) *
mapGen.terrainData.getMapDimension().y) / 2f;

            for (int i = 0; i < points.Count; i++)
            {
                  // mesh generation occurs from top left down, thus y values must
be done inversely
                  float sampleX = topLeftX + points[i].x;
                  float sampleY = topLeftZ - points[i].y;
                  int floorNoiseX = Mathf.FloorToInt(points[i].x);
                  int floorNoiseY = Mathf.FloorToInt(points[i].y);

                  Vector3 bottomLeft = new Vector3(floorNoiseX,
heightCurve.Evaluate(mapGen.wholeNoiseMap[floorNoiseX, floorNoiseY]), floorNoiseY);
                  Vector3 topLeft = new Vector3(floorNoiseX,
heightCurve.Evaluate(mapGen.wholeNoiseMap[floorNoiseX, floorNoiseY]), floorNoiseY +
1);
                  Vector3 bottomRight = new Vector3(floorNoiseX + 1,
heightCurve.Evaluate(mapGen.wholeNoiseMap[floorNoiseX, floorNoiseY]), floorNoiseY);
                  Vector3 topRight = new Vector3(floorNoiseX + 1,
heightCurve.Evaluate(mapGen.wholeNoiseMap[floorNoiseX, floorNoiseY]), floorNoiseY +
1);


                  float current =
heightCurve.Evaluate(mapGen.wholeNoiseMap[(int)points[i].x, (int)points[i].y]);

                  if (Physics.Raycast(new Vector3(sampleX, 50, sampleY),
Vector3.down, out hit, Mathf.Infinity))
                  {
                        // stores height of terrain point
                        height = hit.point.y;
                  }
                  // only spawns object on grass
                  if (current < mapGen.textureData.layers[4].startHeight && current
> mapGen.textureData.layers[2].startHeight && hit.collider.tag == "Terrain Section")
                  {
                        // spawns an object at the generated point and respective
height
                        GameObject temp =
gameObjectArray[UnityEngine.Random.Range(0, gameObjectArray.Length)];
                        Instantiate(temp, new Vector3(sampleX, height, sampleY),
Quaternion.identity);
                        // rotates object randomly around y - axis (rotation swaps
y and z)
                        temp.transform.Rotate(0, 0, UnityEngine.Random.Range(0,
360));
                  }


            }
            points.Clear();
```

```
        }
```

MapDisplay.cs

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

public class MapDisplay : MonoBehaviour {

    private Renderer textureRenderer;
    private MeshFilter meshFilter;
    private MeshRenderer meshRenderer;

    public void DrawTexture(Texture2D texture)
    {
        textureRenderer.sharedMaterial.mainTexture = texture;
        textureRenderer.transform.localScale = new Vector3(texture.width, 1,
texture.height);
    }

    public void DrawMesh(MeshData meshData)
    {
        meshFilter.sharedMesh = meshData.CreateMesh();

            meshFilter.transform.localScale = Vector3.one *
FindObjectOfType<MapGen>().terrainData.getUniformScale();
    }

}
```

Terrain.shader

```
Shader "Custom/Terrain" {
       Properties{
            testTexture("Texture", 2D) = "white"{}
            testScale("Scale", Float) = 1
       }
       SubShader{
            Tags { "RenderType" = "Opaque" }
            LOD 200

            CGPROGRAM
            // Physically based Standard lighting model, and enable shadows on all
light types
#pragma surface surf Standard fullforwardshadows

            // Use shader model 3.0 target, to get nicer looking lighting
#pragma target 3.0

            const static int maxLayerCount = 8;
const static float epsilon = 1E-4;

int layerCount;
float3 baseColours[maxLayerCount];
float baseStartHeights[maxLayerCount];
```

```
float baseBlends[maxLayerCount];
float baseColourStrength[maxLayerCount];
float baseTextureScales[maxLayerCount];

float minHeight;
float maxHeight;

sampler2D testTexture;
float testScale;

UNITY_DECLARE_TEX2DARRAY(baseTextures;)

		sampler2D _MainTex;

struct Input
{
	float3 worldPos;
	float3 worldNormal;
};

float inverseLerp(float a, float b, float value)
{
	return saturate((value - a) / (b - a));
}

float3 triplanar(float3 worldPos, float scale, float3 blendAxes, int textureIndex)
{
// triplanar mapping prevents stretching of texture in any axis
	float3 scaledWorldPos = worldPos / scale;
	float3 xProjection = UNITY_SAMPLE_TEX2DARRAY(baseTextures,
float3(scaledWorldPos.y, scaledWorldPos.z, textureIndex)) * blendAxes.x;
	float3 yProjection = UNITY_SAMPLE_TEX2DARRAY(baseTextures,
float3(scaledWorldPos.x, scaledWorldPos.z, textureIndex)) * blendAxes.y;
	float3 zProjection = UNITY_SAMPLE_TEX2DARRAY(baseTextures,
float3(scaledWorldPos.x, scaledWorldPos.y, textureIndex)) * blendAxes.z;
	return xProjection + yProjection + zProjection;
}

void surf(Input IN, inout SurfaceOutputStandard o)
{
	float heightPercent = inverseLerp(minHeight, maxHeight, IN.worldPos.y);
	float3 blendAxes = abs(IN.worldNormal);
	blendAxes /= blendAxes.x + blendAxes.y + blendAxes.z;
	for (int i = 0; i < layerCount; i++)
	{
		float drawStrength = inverseLerp(-baseBlends[i] / 2 - epsilon,
baseBlends[i] / 2, heightPercent - baseStartHeights[i]);

		float3 baseColour = baseColours[i] * baseColourStrength[i];
		float3 textureColor = triplanar(IN.worldPos, baseTextureScales[i],
blendAxes, i) * (1 - baseColourStrength[i]);

		o.Albedo = o.Albedo * (1 - drawStrength) + (baseColour + textureColor) *
drawStrength;
	}

}
ENDCG
```

```
        }
        FallBack "Diffuse"
}
```

Item Functionality

Item.cs

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

[Serializable]
public class Item
{
// mainly data storage class
        [SerializeField]
        private string displayName;
        private string description;
        [SerializeField]
        public GameObject itemObject;
        [SerializeField]
        private bool has = false;
        private float minRadius;
        private float maxRadius;
        public Item(string _name, string _description, bool _has, GameObject
_itemObject)
        {
                has = _has;
                displayName = _name;
                description = _description;
                itemObject = _itemObject;
        }
        // constructor chaining
        public Item(string _name, string _description, bool _has, GameObject
_itemObject, float _minRadius, float _maxRadius) : this(_name, _description, _has,
_itemObject)
        {
                minRadius = _minRadius;
                maxRadius = _maxRadius;
        }
        public string getDescription()
        {
                return description;
        }
        public string getDisplayName()
        {
                return displayName;
        }
        public bool getHas()
        {
                return has;
        }
        public void setHas(bool _has)
```

```
	{
		this.has = _has;
	}
	public float getMinRadius()
	{
		return minRadius;
	}
	public float getMaxRadius()
	{
		return maxRadius;
	}

}
```

IconController.cs

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

public class IconController : MonoBehaviour {

	Quaternion rotation;
	private void Awake()
	{
		transform.rotation = Quaternion.identity;
		transform.Rotate(90, 0, 0);
		rotation = transform.rotation;
	}
	private void LateUpdate()
	{
		transform.rotation = rotation;
	}

}
```

Camera

CameraFollow.cs

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

public class CameraFollow : MonoBehaviour {

	[SerializeField]
	private float cameraMoveSpeed = 120.0f;
	[SerializeField]
	public float targetingMoveSpeed = 0.5f;
	[SerializeField]
	private GameObject cameraFollowObject;
	[SerializeField]
	private float clampAngle = 80.0f;
```

```
	private float mouseX, mouseY;
	private float finalInputX, finalInputZ;
	private float smoothX, smoothY;
	private float rotationY, rotationX = 0;
	[SerializeField]
	private int inversionX;
	[SerializeField]
	private int inversionY;
	private CameraState cameraState = CameraState.Free;
	public enum CameraState
	{
		Target, Free
	}

	private void Awake()
	{
		// Loading camera inversion preferences
		inversionX = PlayerPrefs.GetInt("Camera Inversion X");
		inversionY = PlayerPrefs.GetInt("Camera Inversion Y");
	}
	// Use this for initialization
	void Start () {
		Vector3 rotation = transform.localRotation.eulerAngles;
		rotationY = rotation.y;
		rotationX = rotation.x;
		Cursor.lockState = CursorLockMode.Locked;
		Cursor.visible = false;
	}

	// Update is called once per frame
	void Update () {

		if (Input.GetAxis("Target") > 0.01 || Input.GetKeyDown(KeyCode.Z))
		{
			//Debug.Log("Camera has been focused");
			TargetingRotation();
		}
		else
		{
			// takes in both mouse and controller input
			float controllerInputX = Input.GetAxis("RightStickHorizontal");
			float controllerInputZ = Input.GetAxis("RightStickVertical");
			mouseX = Input.GetAxis("Mouse X");
			mouseY = Input.GetAxis("Mouse Y");
			finalInputX = controllerInputX + mouseX;
			finalInputZ = controllerInputZ + mouseY;

			// mouse movements in x correspond to rotation about y-axis
			// mouse movements in y correspond to rotation about x-axis
			rotationY += inversionX * finalInputX * cameraMoveSpeed *
Time.deltaTime;
			rotationX += -inversionY * finalInputZ * cameraMoveSpeed *
Time.deltaTime;
			// prevents camera from rotating past clamped angle giving
unpredictable rotation
			rotationX = Mathf.Clamp(rotationX, -clampAngle, clampAngle);
			transform.rotation = Quaternion.Euler(rotationX, rotationY, 0);
		}
```

```
	}

	void LateUpdate()
	{

		UpdateCameraBase();
	}
	void UpdateCameraBase()
	{
		// makes sure that camera follows player's movement
		Transform target = cameraFollowObject.transform;
		float step = cameraMoveSpeed * Time.deltaTime;
		transform.position = Vector3.MoveTowards(transform.position,
target.position, step);
	}
	void TargetingRotation()
	{
		// postions camera directly behind player
		Transform target = cameraFollowObject.transform;
		rotationY = target.rotation.eulerAngles.y;
		rotationX = target.rotation.eulerAngles.x;
		float step = targetingMoveSpeed * Time.deltaTime;
		transform.rotation = Quaternion.Lerp(transform.rotation,
target.rotation, step);
	}

}
```

CameraCollision.cs

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

public class CameraCollision : MonoBehaviour
{
	[SerializeField]
	private float minDistance = 0.2f;
	[SerializeField]
	private float maxDistance = 5.0f;
	[SerializeField]
	private float smooth = 10.0f;
	[SerializeField]
	public float targetSmooth = 1000.0f;
	private Vector3 dollyDir;
	private float distance;


	void Awake()
	{
		maxDistance = PlayerPrefs.GetFloat("Camera Distance");
		//Debug.Log("Standard Distance: " + maxDistance);
		dollyDir = transform.localPosition.normalized; // distance from camera
base, normalised to have a magnitude of 1
		distance = transform.localPosition.magnitude; // distance * dollyDir =
distance of the camera from the base
	}
```

```
	// Update is called once per frame
	void Update()
	{
		SetFreePosition();
	}
	public void SetFreePosition()
	{
		Vector3 desiredCameraPosition = transform.parent.TransformPoint(dollyDir
* maxDistance); // the desired position is the closest it can be to the camera base
when moving by the max distance
		RaycastHit hit;

		if (Physics.Linecast(transform.parent.position, desiredCameraPosition,
out hit))
		{
			distance = Mathf.Clamp(hit.distance * 0.9f, minDistance,
maxDistance); // returns the distance from the camera to the hit object reduced by 10%
		}
		else
		{
			distance = maxDistance;
		}
		transform.localPosition = Vector3.Lerp(transform.localPosition, dollyDir
* distance, Time.deltaTime * smooth);
		//Debug.Log("Distance from player: " + distance);
	}

}
```

Player and Enemies

Character.cs

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

public class Character : MonoBehaviour
{
// base class for living entities
	[SerializeField]
	protected Stat health;
	public float getHealth()
	{
		return health.CurrentValue;

	}

}
```

Player.cs

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

```
using System.Linq;
using UnityEditor;

public class Player : Character {

	private GameObject inventoryMenu;
	private Manager manager;
	[SerializeField]
	private GameObject compass;
	[SerializeField]
	private GameObject pathfinder;
	[SerializeField]
	private GameObject enemyFinder;
	[SerializeField]
	private GameObject playerMarker;
	[SerializeField]
	private GameObject goal;
	private MapGen mapGen;
	private Spawner spawner;
	[SerializeField]
	public Dictionary<string, Item> collectables;
	// Use this for initialization
	void Start()
	{
		mapGen = FindObjectOfType<MapGen>();
		spawner = FindObjectOfType<Spawner>();
		float mapConstant = mapGen.mapSectionSize *
(mapGen.terrainData.getMapDimension().x + mapGen.terrainData.getMapDimension().y) / 4;
		// create collectable item instances
		inventoryMenu = GameObject.FindGameObjectWithTag("Inventory Menu");
		manager = FindObjectOfType<Manager>();

		// dictionary to store all collectables, allows for quick lookup on
collisions
		collectables = new Dictionary<string, Item>()
		{
			{ "Compass", new Item("Compass", "The compass will integrate
itself into your minimap. You should now find it easier to orientate yourself.",
false, compass, 0.1f * mapConstant, 0.2f * mapConstant) },
			{ "Pathfinder", new Item("Pathfinder", "When viewing the Pause
Menu (P) , the Pathfinder will suggest a route. This will help you to avoid obstacles
that may slow you down.", false, pathfinder, 0.6f * mapConstant, 0.75f * mapConstant)
},
			{ "Enemy Finder", new Item("Enemy Finder", "When viewing the
Pause Menu (P) , the Enemy Finder will highlight the current location of all enemies
on the map.", false, enemyFinder, 0.2f * mapConstant, 0.65f * mapConstant) },
			{ "Player Marker", new Item("Player Marker", "When viewing the
Pause Menu (P) , the Player Marker will highlight your position on the map.", false,
playerMarker, 0.25f * mapConstant, 0.4f * mapConstant) },
			{ "Collectable", new Item("Goal Collectable", "Well Done! You
have located the goal collectable, this means that you have won!", false, goal, 0.6f *
mapConstant, 0.9f * mapConstant) }
		};
		spawner.SpawnCollectables(collectables);
		collectables["Player Marker"].setHas(true);
		//collectables["Enemy Finder"].setHas(true);
		collectables["Pathfinder"].setHas(true);
	}
```

```
	private void Awake()
	{
		health.Initialise();
	}
	// Update is called once per frame
	void Update () {
		if(health.CurrentValue == 0)
		{
			// checks whether the plaeyr has lost
			manager.DisplayGameOver();
		}
	}
	public void DamagePlayer(float damage)
	{
		health.CurrentValue -= damage;
		//Debug.Log("Lost " + damage + " health");
	}

	// checks whether a player's collision is with one of the collectables, if so,
applys appropriate result
	private void OnTriggerEnter(Collider other)
	{
		string collidedTag = other.gameObject.tag;
		if(collectables.ContainsKey(collidedTag))
		{
			other.gameObject.SetActive(false);
			collectables[collidedTag].setHas(true);
			manager.DisplayPopup(collectables[collidedTag]);
			if(collidedTag == "Collectable")
			{
				// ends game if player has reached goal
				Application.Quit();
			}
			else
			{
				// all other collectables restore health by 40
				health.CurrentValue += 40;
			}
		}
	}

}
```

Enemy.cs

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

public class Enemy : Character {

	private Player pplayer;
	private GameObject player;
	[SerializeField, Range(0,360)]
	// stores the field of view that the enemy can detect player within
	private float fovAngle;
	[SerializeField]
```

```
// stores the distance that the player can be detected within
private float fovDistance;
public float getFOVDistance()
{
	return fovDistance;
}
public void setFOVDistance(float _fovDistance)
{
	fovDistance = _fovDistance;
}
public LayerMask playerMask;
public LayerMask obstacleMask;

// used to represent whether the player can be detected
private bool playerinRange;
private bool playerinAngle;

private Vector3 vectorToPlayer;
private bool playerVisible;
[SerializeField]
private float hoverHeight = 1.2f;
[SerializeField]
private float hoverForce = 65.0f;
[SerializeField]
private float movementSpeed;
[SerializeField]
// Stores the closest distance that the enemy will move towards the player
private float closestApproach = 1f;
[SerializeField]
// How long the enemy will move before changing direction
private float wanderTime;
public GameObject projectile;
private int visibleCount;
private Vector3 playerUpPosition;

private void Start()
{
	// begins search with 0.2s between each
	StartCoroutine("SearchWithDelay", .2f);

}

private void Update()
{

}

public void Awake()
{
	pplayer = FindObjectOfType<Player>();
	player = pplayer.gameObject;
}

private void FixedUpdate()
{
	Rigidbody rigidbody = GetComponent<Rigidbody>();
	Ray ray = new Ray(transform.position, Vector3.down);
	RaycastHit hit;
```

```
            playerUpPosition = new Vector3(player.transform.position.x,
player.transform.position.y + 1f, player.transform.position.z);

            if (!playerVisible)
            {
                  // If player isn't visible, walks in random direction for a few
seconds
                  if (wanderTime > 0)
                  {
                        this.transform.Translate(Vector3.forward * movementSpeed);
                        wanderTime -= Time.deltaTime;
                  }
                  else
                  {
                        //Debug.Log("New Patrol");
                        wanderTime = Random.Range(3.0f, 10.0f);
                        //Debug.Log("Time: " + wanderTime);
                        int rangle = Random.Range(30, 360);
                        transform.eulerAngles = new Vector3(0, rangle, 0);
                        //Debug.Log("Angle: " + rangle);
                  }

            }
            else
            {
                  // moves towards player when detected
                  wanderTime = 0;
                  transform.LookAt(player.transform);
                  Vector3 desiredPosition = playerUpPosition- closestApproach *
vectorToPlayer.normalized;
                  transform.position = Vector3.MoveTowards(transform.position,
desiredPosition, Time.deltaTime);
            }
            if (Physics.Raycast(ray, out hit, hoverHeight))
            {
                  // allows enemy to hover
                  float heightPercent = (hoverHeight - hit.distance) / hoverHeight;
                  Vector3 appliedHoverForce = Vector3.up * heightPercent *
hoverForce;
                  rigidbody.AddForce(appliedHoverForce, ForceMode.Acceleration);
            }
            transform.position += 0.03f * new
Vector3(0,Mathf.Cos(Time.deltaTime),0);
      }

      void CheckPlayerRange()
      {
            // checks if player is within minimum sight distance
            if(Vector3.Magnitude(vectorToPlayer) < fovDistance )
            {
                  playerinRange = true;
            }
            else
            {
                  playerinRange = false;
            }
      }
```

```
	void CheckPlayerAngle()
	{
		// checks whether the player is within the field of view angle
		if(Vector3.Angle(transform.forward, vectorToPlayer.normalized) < 
fovAngle / 2)
		{
			playerinAngle = true;
		}
		else
		{
			playerinAngle = false;
		}
	}

	IEnumerator SearchWithDelay(float delay)
	{
		while(true)
		{
			yield return new WaitForSeconds(delay);
			Detect();
		}
	}

	public void Detect ()
	{
		vectorToPlayer = player.transform.position - this.transform.position;
		CheckPlayerRange();
		CheckPlayerAngle();
		if (playerinAngle && playerinRange)
		{
			// player is visible
			if(!Physics.Raycast(transform.position, 
vectorToPlayer.normalized, Vector3.Magnitude(vectorToPlayer), obstacleMask))
			{
				playerVisible = true;
				//Debug.Log("Detected");
				visibleCount++;
			}
			else
			{
				visibleCount = 0;
				//Debug.Log("Now Undetected");
				playerVisible = false;
			}
		}
		else
		{
			// player isn't visible
			visibleCount = 0;
			playerVisible = false;
		}
		if(visibleCount > 0 && visibleCount % 5 == 0)
		{
			// fires projectile every 2 seconds player is visible for
			Fire();
		}
	}
```

```
	public void Fire()
	{
		// creates projectile
		Instantiate(projectile, transform.position, Quaternion.identity);
	}

	public Vector3 DirectionFromAngle(float angle, bool angleIsGlobal)
	{
		if(!angleIsGlobal)
		{
			angle += transform.eulerAngles.y;
		}
		return new Vector3(Mathf.Sin(angle * Mathf.Deg2Rad), 0, Mathf.Cos(angle*
Mathf.Deg2Rad));
	}

}
```

Projectile.cs

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

public class Projectile : MonoBehaviour {

	[SerializeField]
	private float speed;
	private Player player;
	[SerializeField]
	private GameObject hitEffect;
	private Vector3 target;
	private Vector3 ini;
	private Vector3 vToTarget;


	private void Start()
	{
		player = FindObjectOfType<Player>();
		target = player.transform.position; // stores players position upon
spawn
		ini = transform.position; // stores position spawned at
		target.y += 1f;
		vToTarget = target - ini;
	}

	private void Update()
	{
		transform.position = Vector3.MoveTowards(transform.position, ini + 2 *
vToTarget, speed * Time.deltaTime);

		if(ini -  transform.position == 2 * (ini - target))
		{
			// Destroys projectile once it reaches twice the distance it was
aiming at
			DestroyProjectile();
		}
```

```
	}

	private void OnTriggerEnter(Collider collided)
	{
		// destroys projectile when it collides with another object
		if(collided.CompareTag("Player"))
		{
			// Damages player when hit by projectile
			player.DamagePlayer(5);
		}
		if(!collided.CompareTag("Enemy"))
		{
			// Prevents projectile from being destroyed by hitting enemy
itself
			DestroyProjectile();
		}
	}

	void DestroyProjectile()
	{
		Instantiate(hitEffect, transform.position, Quaternion.identity);
		Destroy(gameObject);
	}

}
```

Pathfinding

Coordinate.cs

```
//Prevents issue of reference equality being used in int arrays
using UnityEngine;

public class Coordinate
{

	private int x;
	public int getX() // stores positive x
	{
		return x;
	}

	private int y;
	public int getY() // stores positive y
	{
		return y;
	}


	private float height;
	public float getHeight() // stores the height of the terrain at coordinate
	{
		return height;
	}
	public void setHeight(float _height)
```

```
        {
                height = _height;
        }

        private bool onFrontier;
        public bool OnFrontier()
        {
                return onFrontier;
        }
        public void setOnFrontier(bool _onFrontier)
        {
                onFrontier = _onFrontier;
        }

        private bool closed;
        public bool Closed() // stores whether node has been visited before
        {
                return closed;
        }
        public void setClosed(bool _closed)
        {
                closed = _closed;
        }

        private float cost;
        public float getCost() // stores the node's cost
        {
                return cost;
        }
        public void setCost(float _cost)
        {
                cost = _cost;
        }

        private float heuristic;
        public float getHeuristic()
        {
                return heuristic;
        }
        public void setHeuristic(float _heuristic)
        {
                heuristic = _heuristic;
        }

        private float enemyCost;
        public float getEnemyCost() // stores the node's value from enemy influence
        {
                return enemyCost;
        }
        public void setEnemyCost(float enemyCost)
        {
                this.enemyCost = enemyCost;
        }

        // marks the node that this was reached from, such that the path can be
backtracked
        public Coordinate cameFrom;
```

```
        public Coordinate(int x, int y, float height)
        {
                this.x = x;
                this.y = y;
                this.height = height;
                onFrontier = false;
                closed = false;
                cost = float.MaxValue;
                //Indicator value that heuristic has not been set
                heuristic = -1f;
                cameFrom = null;
                enemyCost = 0;
        }

        public Coordinate(Vector2 pos, float height) : this((int)pos.x, (int)pos.y,
height) { }

        public void Reset()
        {
                // Resets all values as pathfinding may occur multiple times
                onFrontier = false;
                closed = false;
                cost = float.MaxValue;
                heuristic = -1;
                cameFrom = null;
                enemyCost = 0;
        }

        public float DistanceTo(Coordinate p, float[,] heightMap)
        {
                // Finds the distance between two nodes in 3D
                int xSum = x - p.x;
                int ySum = y - p.y;
                float zSum = heightMap[x, y] - heightMap[p.x, p.y];
                return Mathf.Sqrt(xSum * xSum + ySum * ySum + zSum * zSum);
        }

        public float DistanceTo2D(Coordinate p)
        {
                // Finds the planar distance between two nodes
                int xSum = x - p.x;
                int ySum = y - p.y;
                return Mathf.Sqrt(xSum * xSum + ySum * ySum);
        }

        public Vector3 ToVector3()
        {
                return new Vector3(x, height, y);
        }

}
```

Pathfinding.cs

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

```
public class Pathfinder
{
	// stores square root of two so it needn't be calculated 1000s of times
	private const float sqrt2 = 1.41421356237f;
	// stores of the vertices in the map, used as the graph
	private Coordinate[,] graph;
	// stores the height of all the vertices in the world
	private float[,] heightMap;
	private float heuristicCoeff = 0.5f;
	private float costCoeff = 30;    //standard 30
	private float heightCostCoeff = 1.1f;  //standard 0.5
	private float enemyCostCoeff = 30f;
	private float seaLevel;
	float maxtotalcost = 0;
	float averagecost = 0;
	int n = 0;
	public Pathfinder(float[,] heightMap, float seaLevel)
	{
		this.seaLevel = seaLevel;
		this.heightMap = heightMap;

		// creates the graph using heightmap data
		graph = new Coordinate[heightMap.GetLength(0), heightMap.GetLength(1)];
		for (int y = 0; y < heightMap.GetLength(1); y++)
		{
			for (int x = 0; x < heightMap.GetLength(0); x++)
			{
				graph[x,y] = new Coordinate(x, y, heightMap[x, y]);
			}
		}
	}
	public List<Vector3> AStar(Vector2 playerPos, Vector2 target, bool
hasEnemyFinder)
	{
		var timer = new System.Diagnostics.Stopwatch();
		timer.Start();

		// finds the node that the goal collectable is located at
		Coordinate goal = graph[(int)target.x + heightMap.GetLength(0) / 2,
(int)target.y + heightMap.GetLength(1) / 2];

		if (hasEnemyFinder)
		{
			Enemy enemyPrefab = Object.FindObjectOfType<Enemy>();
			GameObject[] enemies =
GameObject.FindGameObjectsWithTag("Enemy");
			foreach(GameObject enemy in enemies)
			{
				// prevent enemies that span the goal from having an
influence
				if(target.x >= enemy.transform.position.x -
enemyPrefab.getFOVDistance() && target.x <= enemy.transform.position.x +
enemyPrefab.getFOVDistance()
					&& target.y >= enemy.transform.position.y -
enemyPrefab.getFOVDistance()
					&& target.y <= enemy.transform.position.y +
enemyPrefab.getFOVDistance())
				{
```

```
                                continue;
                            }
                            // will adjust path by changing costs of nodes near
enemies
                            // considers any nodes within twice of the enemies fov
                            Vector2 enemyPosition = new
Vector2((int)enemy.transform.position.x, (int)enemy.transform.position.z);
                            for (int y = (int)-enemyPrefab.getFOVDistance(); y <=
(int)enemyPrefab.getFOVDistance(); y++)
                            {
                                for (int x = (int)-enemyPrefab.getFOVDistance(); x
<= (int)enemyPrefab.getFOVDistance(); x++)
                                {
                                    float displacement = Mathf.Sqrt(x * x + y *
y);
                                    int indexX = (int)(x + enemyPosition.x +
heightMap.GetLength(0) / 2);
                                    int indexY = (int)(y + enemyPosition.y +
heightMap.GetLength(1) / 2);
                                    // testing fix to prevent indexing error
                                    if (indexX >= 0 && indexX <=
graph.GetLength(0) && indexY >= 0 && indexY <= graph.GetLength(1))
                                    {
                                        graph[indexX,
indexY].setEnemyCost((enemyPrefab.getFOVDistance() - displacement) /
enemyPrefab.getFOVDistance() * enemyCostCoeff + enemyCostCoeff);
                                        //enemyPrefab.getFOVDistance() /
displacement
                                    }

                                }
                            }
                        }
                    }
                }



                // finds the node that the player is located at
                Coordinate current = graph[(int)playerPos.x + heightMap.GetLength(0) /
2, (int)playerPos.y + heightMap.GetLength(1) / 2];

                // creates the frontier, where all nodes to be explored from are stored
                PriorityQueue frontier = new PriorityQueue();

                // stores nodes that have already been visited
                List<Coordinate> used = new List<Coordinate>();

                //adds the player's node to the frontier
                //new nodes are found from neighbours of frontier nodes
                frontier.EnQueue(current);

                // marks that the player's node has been visited
                used.Add(current);
                current.setOnFrontier(true);
                current.setCost(0);

                while (frontier.Count() > 0)
                {
```

```
                        // removes the current node from the frontier as it has been
explored
                        current = frontier.DeQueue();
                        if (current == goal)
                        {
                                // stops algorithm if goal has been reached
                                break;
                        }
                        current.setOnFrontier(false);
                        current.setClosed(true);
                        // double for loop finds all of the neighbouring nodes, maximum 8
                        for (int y = Mathf.Max(0, current.getY() - 1); y <
Mathf.Min(heightMap.GetLength(1), current.getY() + 2); y++)
                        {
                                for (int x = Mathf.Max(0, current.getX() - 1); x <
Mathf.Min(heightMap.GetLength(0), current.getX() + 2); x++)
                                {
                                        // creates a temporary copy of a neighbouring node
                                        Coordinate temp = graph[x, y];
                                        float newcost = 0f;
                                        if (current == temp)
                                        {
                                                // moves on to next node if the temporary
node is the current node
                                                continue;
                                        }
                                        if (!temp.OnFrontier() && !temp.Closed())
                                        {
                                                //Node has not been seen before, thus can be
added to frontier
                                                temp.setCost(current.getCost() +
MovementCost(current, temp, goal, hasEnemyFinder));
                                                temp.setHeuristic(HeuristicCost(temp, goal));
                                                frontier.EnQueue(temp);
                                                used.Add(temp);
                                                temp.setOnFrontier(true);
                                                // marks that the temporary node was found by
moving from the current node
                                                temp.cameFrom = current;
                                                //using assignment returning the value that
is assigned
                                        }
                                        else if (temp.OnFrontier() && temp.getCost() >
(newcost = current.getCost() + MovementCost(current, temp, goal, hasEnemyFinder)))
                                        {
                                                //Node has been seen before but there is a
faster way of reaching it
                                                frontier.UpdatePriority(temp, newcost);
                                                temp.cameFrom = current;
                                        }
                                }
                        }
                }
                List<Vector3> path = new List<Vector3>();
                while (current.cameFrom != null)
                {
                        path.Add(current.ToVector3());
                        current = current.cameFrom;
```

```
                }
                path.Reverse();
                foreach (Coordinate c in used)
                {
                        c.Reset();
                }
                Debug.Log("Time with Priority Queue is " + timer.ElapsedTicks);
                Debug.Log("Max: " + maxtotalcost);
                Debug.Log("Average: " + averagecost);
                averagecost = 0;
                n = 0;
                timer.Stop();
                return path;
        }

        public float MovementCost(Coordinate from, Coordinate to, Coordinate goal, bool
hasEnemyFinder)
        {
                //Calculates the movement cost between two points based on
distance/incline
                //Need to not punish downhill as much as uphill
                // Need to punish moving in the FOV of enemies
                float distance = Mathf.Abs(from.getX() - to.getX()) +
Mathf.Abs(from.getY() - to.getY()) == 1 ? 1 : sqrt2;
                float cost = to.getHeight() > from.getHeight() ?
Mathf.Abs(to.getHeight() - from.getHeight()) / distance : 0.6f *
Mathf.Abs(to.getHeight() - from.getHeight()) / distance;
                if(hasEnemyFinder)
                {
                cost = cost + (to.getEnemyCost() + from.getEnemyCost()) / 2;
                }
                cost *= costCoeff;
                cost += 1;
                cost += heightCostCoeff * Mathf.Max(goal.getHeight() - to.getHeight(),
1);
                //float totalcost = tan * tan * distance * costCoeff; //+ (to.height -
goal.height) * heightCostCoeff;
                //maxtotalcost = totalcost > maxtotalcost ? totalcost : maxtotalcost;
                averagecost = (averagecost * n + cost) / (float)(++n);
                return cost;

        }

        public float HeuristicCost(Coordinate from, Coordinate to)
        {
                return from.DistanceTo2D(to) * heuristicCoeff;
        }

}


public class PriorityQueue
{
        List<Coordinate> queue;

        public PriorityQueue()
        {
```

```
            queue = new List<Coordinate>();
        }

        public void EnQueue(Coordinate insert)
        {
            // adds a coordinate to the queue, using the priority of the node
            if (queue.Count == 0)
            {
                // if the queue is empty then the node can just be added
                queue.Add(insert);
                return;
            }
            int midpoint = 0;
            // priority is determined by the sum of the cost and heuristic
            int insertPriority = (int)(insert.getCost() + insert.getHeuristic());
            int lowerBound = 0;
            int upperBound = queue.Count - 1;
            while (upperBound > lowerBound)
            {
                // binary insertion
                midpoint = (upperBound + lowerBound) / 2;
                if (insertPriority > (int)(queue[midpoint].getCost() +
queue[midpoint].getHeuristic()))
                {
                    // removes lower section from consideration for next
iteration
                    lowerBound = midpoint + 1;
                }
                else if (insertPriority < (int)(queue[midpoint].getCost() +
queue[midpoint].getHeuristic()))
                {
                    // removes upper section from consideration for next
iteration
                    upperBound = midpoint - 1;
                }
                else
                {
                    // correct position has been found
                    queue.Insert(midpoint + 1, insert);
                    return;
                }
            }
            queue.Insert(insertPriority > (int)(queue[lowerBound].getCost() +
queue[lowerBound].getHeuristic()) ? lowerBound + 1 : lowerBound, insert);
        }
        public Coordinate DeQueue()
        {
            // standard dequeue method
            Coordinate front = queue[0];
            queue.RemoveAt(0);
            return front;
        }
        public void UpdatePriority(Coordinate subject, float cost)
        {
            // a faster way of reaching the subject node has been found, so its
position in the queue must be changed
            int midpoint = 0;
            int subjectPriority = (int)(subject.getCost() + subject.getHeuristic());
```

```
            int lowerBound = 0;
            int upperBound = queue.Count - 1;
            while (upperBound > lowerBound)
            {
                  // binary search
                  midpoint = (upperBound + lowerBound) / 2;
                  int midPointPriority = (int)(queue[midpoint].getCost() +
queue[midpoint].getHeuristic());
                  if (midPointPriority == subjectPriority)
                  {
                        // many nodes may have the same priority - especially from
rounding
                        // meaning that the subject could be either side of the
calculated midpoint
                        int target = CheckSide(subject, midpoint, -1,
subjectPriority);
                        // target stores the index in the queue where the subject
was previously found
                        if (target == -1)
                        {
                              // sign of third parameter of checkside indicates
direction
                              target = CheckSide(subject, midpoint, 1,
subjectPriority);
                              if (target == -1)
                              {
                                    if (!queue.Contains(subject))
                                    {
                                          throw new System.Exception("It's
actually not there");
                                    }
                                    throw new System.Exception("Target was not
found");
                              }
                        }
                        // removes subject from its previous location
                        queue.RemoveAt(target);
                        subject.setCost(cost);
                        // adds subject to the queue, with new position to be
found
                        EnQueue(subject);
                        return;
                  }
                  else if (subjectPriority > midPointPriority)
                  {
                        // removes lower section from consideration for next
iteration
                        lowerBound = midpoint + 1;
                  }
                  else if (subjectPriority < midPointPriority)
                  {
                        // removes upper section from next iteration
                        upperBound = midpoint - 1;
                  }
            }
            // reintroduces subject to queue
            queue.RemoveAt(lowerBound);
            subject.setCost(cost);
```

```
                EnQueue(subject);
        }



        public void UpdatePriority2(Coordinate subject, float cost)
        {
                queue.Remove(subject);
                subject.setCost(cost);
                EnQueue(subject);
        }


        public int CheckSide(Coordinate subject, int midpoint, int direction, int
subjectPriority)
        {
                // moves from a point of equal priority with the subject until priority
changes
                int target = midpoint;

                while (target >= 0 && target < queue.Count && subjectPriority ==
(int)(queue[target].getCost() + queue[target].getHeuristic()))
                {
                        if (subject == queue[target])
                        {
                                return target;
                        }
                        target += direction;
                }
                // -1 marks that the subject isn't found on the side of the midpoint
checked
                return -1;
        }


        public int Count()
        {
                return queue.Count;
        }


}
```

MinimapPathManager.cs

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

public class MinimapPathManager : MonoBehaviour{

        private GameObject collectable;
        private MapGen mapGen;
        [HideInInspector]
        public Texture2D minimapOverlay;
        [SerializeField]
```

```
	private RawImage rawImage;
	private List<Vector3> path;
	public Pathfinder pathfinder;


	private void Start()
	{
		mapGen = FindObjectOfType<MapGen>();
		pathfinder = new Pathfinder(mapGen.realMap,
mapGen.textureData.layers[1].startHeight *
mapGen.terrainData.getMeshHeightMultiplier());
		//minimapOverlay = new Texture2D(720, 720);
		minimapOverlay = new
Texture2D((int)mapGen.terrainData.getMapDimension().x * mapGen.mapSectionSize,
(int)mapGen.terrainData.getMapDimension().y * mapGen.mapSectionSize);
		rawImage.texture = minimapOverlay;
	}
	public void UpdatePath(GameObject player, bool hasPathfinder, bool
hasEnemyFinder, GameObject[] enemies)
	{
		collectable = GameObject.FindGameObjectWithTag("Collectable");
		// removes previous values on texture
		ClearMap();
		if(hasPathfinder)
		{
			// finds and draws path if player has item
			path = pathfinder.AStar(new Vector2(player.transform.position.x,
player.transform.position.z),
new Vector2(collectable.transform.position.x, collectable.transform.position.z),
hasEnemyFinder);
			DrawPath();
		}
		// applys path to minimap
		minimapOverlay.Apply();
	}
	public void DrawPath()
	{
		foreach (Vector3 p in path)
		{
			for (int y = (int)p.z - 1; y <= (int)p.z + 1; y++)
			{
				for (int x = (int)p.x - 1; x <= (int)p.x + 1; x++)
				{
					// colours all nodes that are found on the path
					// colours the immediate neighbours of each node
for thickness
					minimapOverlay.SetPixel(x, y, Color.cyan);
				}
			}
		}
	}
	public void ClearMap()
	{
		for (int i = 0; i < mapGen.terrainData.getMapDimension().x *
mapGen.mapSectionSize; i++)
		{
			for (int j = 0; j < mapGen.terrainData.getMapDimension().y *
mapGen.mapSectionSize; j++)
```

```
                        {
                                // resets texture
                                minimapOverlay.SetPixel(i, j, Color.clear);
                        }
                }
        }

}
```

UI

MainMenu.cs

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

public class MainMenu : MonoBehaviour {

        [SerializeField]
        private Scene gameScene;

        public void PlayGame()
        {
                SceneManager.LoadScene("GameScene");
        }
        public void QuitGame()
        {
                Application.Quit();
        }

}
```

SettingsMenu.cs

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

public class SettingsMenu : MonoBehaviour
{
        [SerializeField]
        private GameObject distanceSlider;
        [SerializeField]
        private GameObject toggleInversionXOn;
        [SerializeField]
        private GameObject toggleInversionXOff;
        [SerializeField]
        private GameObject toggleInversionYOn;
        [SerializeField]
        private GameObject toggleInversionYOff;

        public void Start()
        {
```

```
                Load();
        }
        public void Apply()
        {
                // saves changes made in UI
                PlayerPrefs.SetFloat("Camera Distance",
distanceSlider.GetComponent<Slider>().value);
                if(toggleInversionXOn.activeInHierarchy)
                {
                        PlayerPrefs.SetInt("Camera Inversion X", -1);
                }
                else
                {
                        PlayerPrefs.SetInt("Camera Inversion X", 1);
                }
                if (toggleInversionYOn.activeInHierarchy)
                {
                        PlayerPrefs.SetInt("Camera Inversion Y", -1);
                }
                else
                {
                        PlayerPrefs.SetInt("Camera Inversion Y", 1);
                }
        }
        public void Load()
        {
                // adjusts UI based on saved values
                distanceSlider.GetComponent<Slider>().value =
PlayerPrefs.GetFloat("Camera Distance");
                if(PlayerPrefs.GetInt("Camera Inversion X") == -1)
                {
                        toggleInversionXOn.SetActive(true);
                        toggleInversionXOff.SetActive(false);
                }
                else
                {
                        toggleInversionXOn.SetActive(false);
                        toggleInversionXOff.SetActive(true);
                }
                if (PlayerPrefs.GetInt("Camera Inversion Y") == -1)
                {
                        toggleInversionYOn.SetActive(true);
                        toggleInversionYOff.SetActive(false);
                }
                else
                {
                        toggleInversionYOn.SetActive(false);
                        toggleInversionYOff.SetActive(true);
                }
        }

}
```

Minimap.cs

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

```
using UnityEngine;

public class Minimap : MonoBehaviour
{
	[SerializeField]
	private Transform Player;
	[SerializeField]
	private Transform MainCamera;
	[SerializeField]
	private GameObject North;

	private void LateUpdate()
	{
		Vector3 newPosition = Player.position;
		newPosition.y = transform.position.y;
		transform.position = newPosition;
		// rotates minimap with camera
		transform.rotation = Quaternion.Euler(90f, MainCamera.eulerAngles.y, 0);
		// rotates north marker around minimap circle
		North.transform.localPosition = new Vector3(Mathf.Sin(-
MainCamera.eulerAngles.y * Mathf.PI / 180) * 50, Mathf.Cos(MainCamera.eulerAngles.y *
Mathf.PI / 180) * 50, 0);
	}

}
```

Marker.cs

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

public class Marker : MonoBehaviour {

	public Transform player;
	[SerializeField]
	private Transform mainCamera;
	private Vector3 desiredEuler;
	private Vector3 playerEuler;
	private Vector3 cameraEuler;

	// Update is called once per frame
	void LateUpdate () {
		cameraEuler = mainCamera.rotation.eulerAngles;
		playerEuler = player.rotation.eulerAngles;

		desiredEuler = new Vector3(0, 0, 57.5f) - new Vector3(0, 0,
playerEuler.y) + new Vector3(0,0, cameraEuler.y);
		transform.rotation = Quaternion.Euler(desiredEuler);

	}

}

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

```
public class MarkerBob : MonoBehaviour {

	private Vector3 ini;
	private float totalTime = 0;
	[SerializeField]
	private float speed;
	[SerializeField]
	private float amplitude;

	public void MoveCosine ()
	{
		transform.localPosition = new Vector3(ini.x, 10 + ini.y + amplitude *
Mathf.Abs(Mathf.Cos(totalTime * speed)), ini.z);
	}
	public void SetPosition(Vector3 playerPos, int mapSize)
	{
		// finds the player's position with regard to map
		float percentX = playerPos.x / mapSize;
		float percentY = playerPos.z / mapSize;
		transform.localPosition = new Vector3(percentX * 360, percentY * 360,
0);
		ini = transform.localPosition;
		StartCoroutine("Oscillate", .05f);
	}
	IEnumerator Oscillate(float delay)
	{
		while(gameObject.activeInHierarchy)
		{
			// allows co-routines to run whilst timescale is 0
			yield return new WaitForSecondsRealtime(delay);
			totalTime += delay;
			MoveCosine();
		}
	}

}
```

General

Manager.cs

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

public class Manager : MonoBehaviour {

	[SerializeField]
	private GameObject player;
	[SerializeField]
	private GameObject minimapCamera;
	[SerializeField]
	private GameObject menuMinimapCamera;
```

```
	private Player pPlayer;
	[SerializeField]
	private GameObject inventoryMenu;
	private MapGen mapGen;
	private GameObject collectable;
	[SerializeField]
	private GameObject minimap;
	[SerializeField]
	private GameObject north;
	[SerializeField]
	private GameObject pauseMarker;
	private MinimapPathManager minimapPathManager;
	[SerializeField]
	private GameObject popup;
	[SerializeField]
	private GameObject popupTitle;
	[SerializeField]
	private GameObject popupText;
	[SerializeField]
	private Button popupButton;
	private Text popupTitleText;
	private GameObject[] enemies;
	private Text popupTextText;

	private bool paused = false;
	void Start () {
		popup.SetActive(false);
		mapGen = FindObjectOfType<MapGen>();
		pPlayer = FindObjectOfType<Player>();
		menuMinimapCamera.GetComponent<Camera>().orthographicSize =
mapGen.terrainData.getMapDimension().x * 240 / 2;
	}

	// Update is called once per frame
	void Update ()
	{
		if (Input.GetKeyDown(KeyCode.P))
		{

	menuMinimapCamera.SetActive(!menuMinimapCamera.activeInHierarchy);
			paused = !paused;
			// shows inventory menu
			inventoryMenu.SetActive(!inventoryMenu.activeInHierarchy);
			// disables standard minimap
			minimap.SetActive(!minimap.activeInHierarchy);
			minimapPathManager = GetComponent<MinimapPathManager>();
			if (paused)
			{

	pauseMarker.GetComponent<MarkerBob>().SetPosition(pPlayer.transform.position,
mapGen.wholeNoiseMap.GetLength(0));
				enemies = GameObject.FindGameObjectsWithTag("Enemy");
				// pauses game time
				Time.timeScale = 0;
				// updates pause minimap
				minimapPathManager.UpdatePath(player,
pPlayer.collectables["Pathfinder"].getHas(), pPlayer.collectables["Enemy
Finder"].getHas(), enemies);
```

```
                }
                else
                {
                        // resumes game on every pause button press
                        Time.timeScale = 1;
                }
        }
        if(pPlayer.collectables["Compass"].getHas())
        {
                // displays compass on minimap if player has the collectable
                north.SetActive(true);
        }
        else
        {
                north.SetActive(false);
        }
        if (pPlayer.collectables["Enemy Finder"].getHas())
        {
                //bitwise shift to change culling of minimap camera
                menuMinimapCamera.GetComponent<Camera>().cullingMask |= (1 <<
14);
                minimapCamera.GetComponent<Camera>().cullingMask |= (1 << 14);
        }
        else
        {
                menuMinimapCamera.GetComponent<Camera>().cullingMask &= ~(1 <<
14);
                minimapCamera.GetComponent<Camera>().cullingMask &= ~(1 << 14);
        }
        if (pPlayer.collectables["Player Marker"].getHas())
        {
                pauseMarker.SetActive(true); // introduces player marker during
pause minimap
        }
        else
        {
                pauseMarker.SetActive(false);
        }
    }
    public void DisplayPopup(Item item)
    {
        // displays collectable information when found in UI pop-up
        popupTitleText = popupTitle.GetComponent<Text>();
        popupTextText = popupText.GetComponent<Text>();
        popupTitleText.text = "You found the " + item.getDisplayName() + "!";
        popupTextText.text = item.getDescription();
        popup.SetActive(true);
        popupButton.Select();
        Time.timeScale = 0;
    }
    public void DisplayGameOver()
    {
        popupTitleText = popupTitle.GetComponent<Text>();
        popupTextText = popupText.GetComponent<Text>();
        popupTitleText.text = "Game Over";
        popupTextText.text = "Your health has fallen to 0.                Make
sure to avoid engaging enemies next time!";
```

```
                popup.SetActive(true);
                popupButton.Select();
                Time.timeScale = 0;
        }
        public void HidePopup()
        {
                if(pPlayer.getHealth() == 0)
                {
                        EditorApplication.isPlaying = false;
                        Application.Quit();
                }
                popup.SetActive(false);
                Time.timeScale = 1;
        }

}
```

Stat.cs

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

[Serializable]
public class Stat
{
// storage class for UI data
        [SerializeField]
        private Bar bar;
        [SerializeField]
        private float maxValue;
        [SerializeField]
        private float currentValue;

        public float CurrentValue
        {
                get
                {
                        return currentValue;
                }

                set
                {
                        this.currentValue = Mathf.Clamp(value, 0, maxValue);
                        bar.Value = currentValue;
                }
        }

        public float MaxValue
        {
                get
                {
                        return maxValue;
                }

                set
```

```
                {
                        this.maxValue = value;
                        bar.MaxValue = value;
                }
        }
        public void Initialise()
        {
                this.MaxValue = maxValue;
                this.CurrentValue = currentValue;
        }
}
```

Bar.cs

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

public class Bar : MonoBehaviour {

        private float fillAmount;
        [SerializeField]
        private Image content;
        [SerializeField]
        private Text valueText;
        [SerializeField]
        private float lerpSpeed;
        public float MaxValue { get; set; }
        public float Value
        {
                set
                {
                        valueText.text = value.ToString();
                        fillAmount = ValuetoBar(value, MaxValue);
                }
        }

        // Use this for initialization
        void Start () {

        }

        // Update is called once per frame
        void Update () {

                UpdateBar();
        }

        private void UpdateBar()
        {
                // fills bar based on player's health percentage
                if (fillAmount != content.fillAmount)
                {
                        content.fillAmount = Mathf.Lerp(content.fillAmount, fillAmount,
Time.deltaTime * lerpSpeed);
                }
```

```
	}
	private float ValuetoBar(float currentValue, float maxValue)
	{
		return Mathf.InverseLerp(0, maxValue, currentValue);
	}

}
```

Rotator.cs

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

public class Rotator : MonoBehaviour {

		// Use this for initialization
	void Start () {

	}

	// Update is called once per frame
	void Update()
	{
		transform.Rotate(new Vector3(15, 30, 45) * Time.deltaTime);
	}


}
```

TreeBehaviour.cs

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

public class TreeBehaviour : MonoBehaviour {

	private GameObject player;
	private Player pPlayer;
	private MapGen mapGen;
	// Use this for initialization
	void Start () {
		pPlayer = FindObjectOfType<Player>();
		player = pPlayer.gameObject;
		mapGen = FindObjectOfType<MapGen>();
	}

	// Update is called once per frame
	void Update ()
	{
		if(Vector3.SqrMagnitude(transform.position - player.transform.position)
< mapGen.getTreeDistanceThreshold() * mapGen.getTreeDistanceThreshold())
		{
			this.transform.localScale = new Vector3(1, 1, 1);
		}
```

```
            else
            {
                this.transform.localScale = new Vector3(0, 0, 0);
            }
        }

}
```

Testing and Evaluation

Default Noise is a parameter that I will use to specify input data that is consistent between tests. It consists of:

- Noise Scale: 400
- Octaves: 7
- Persistence: 0.45
- Lacunarity: 2.2
- Seed: 80
- Offset X: 0
- Offset Y: 0
- UseFalloff: True
- FalloffFunction: Standard

My other general parameter is Default Terrain, this consists of:

- Height Multiplier: 80
- FlatShading: False
- Scale: 1
- MapDimensions: 3x3

I will use brackets following either of these parameters to specify input data where one or more fields differ from these default values.

TEST PLAN

Test No	Description	Test Data	Expected Outcome	Actual Outcome	Evidence
1ai)	Test that noise is applied over a defined number of octaves.	Default Noise, Default Terrain	The system will log each time a new octave is iterated through.	The logger showed the octave count used, before outputting during the use of each octave in terrain generation. Passed.	Test 1ai

1ai)	Test that the resulting terrain changes when the number of octaves differs.	Default Noise (1 (Octaves)), Default Terrain Default Noise (7 (Octaves)), Default Terrain	The two resulting terrains will be different. The second terrain will appear more detailed and less smooth than the first.	The first terrain was far less detailed and angular than the second. Passed.	Test 1aii
1b)	Test that amplitude and frequency are altered between octaves.	Default Noise (4 (Octaves)), Default Terrain	The amplitude and frequency will decrease and increase respectively with each successive octave. The current amplitude and frequency are output in the Console log in each octave.	The system log clearly showed that the amplitude and lacunarity were changing with each successive octave, and only changing as many times as octaves were input. Furthermore, the amplitude and persistence were changing by a factor of the input data each time. Passed.	Test 1b
1c)	Test that the terrain is not uniform in its entirety.	Default Noise, Default Terrain	The map uses height values to determine an appropriate texture, varying the appearance of the terrain.	The terrain is quite clearly varied, containing lots of diverse types of land, shown by different colours and textures. Passed	Test 1c
1d)	Test that the terrain is generated as an island.	Default Noise, Default Terrain	The terrain is surrounded by a dark-blue on all sides, to represent the presence of water.	In each instance the terrain is surrounded by a distinct blue border. This blue texture occurs at the lowest points and is indicative of a sea-level. Passed.	Test 1, all instances

1ei)	Test that multiple sizes of terrain can be successfully generated.	Default Noise, Default Terrain Default Noise, Default Terrain (5 (Map Dimensions))	The terrains generated are 9 and 25 sections large respectively. The system log outputs how many terrain sections have been created in each case, showing a clear distinction.	The two samples show maps of different sizes, and there is plethora of examples indicating that the maps created are 3x3 and 5x5 respectively. The system log shows the terrain count for each, being 9 and 25 as expected. Passed.	Test 1ei
1eii)	Test that the game accommodates multiple sizes of terrain in falloff-map application.	Default Noise, Default Terrain Default Noise, Default Terrain (5 (Map Dimensions))	Both terrains are generated as islands. The larger terrain should have a greater landmass than the smaller terrain.	Both terrains are islands, shown by the blue texture around the outside of each. The larger map, does have a greater amount of landmass, including landmass in a section that would be purely water in the 3x3, illustrating that the fall-off map must adjust to accommodate map size. Passed.	Test 1ei
1eiii)	Test that object spawning is adaptable to maps of different sizes.	Default Noise, Default Terrain Default Noise, Default Terrain (5 (Map Dimensions))	Object spawns spread to fill the entire landmass of each size of map.	The object spawns fill the landmass where appropriate in each case. In the 5x5 Objects are spawned in sections further than the dimensions of the 3x3 map, indicating that the object placement must scale to varied map sizes. Passed.	Test 1eii

1eiv)	Test that the player's map display is adaptable to multiple map sizes.	Default Noise, Default Terrain Default Noise, Default Terrain (5 (Map Dimensions))	The player's map display camera is scaled to capture the entire map, such that the map display fits perfectly in the UI panel. The camera used to render the map display will have a different value for its orthographic size attribute.	The player's map display takes up the same UI space no matter the map size. The camera's have different sizes, the 3x3 holding 360 and the 5x5 holding 600. We can see that these values are proportional to the map size. Passed.	Test 1eiii
1f)	Test that there is no visible seam at the borders between terrain meshes.	Default Noise, Default Terrain	Border's between meshes are seamless, such that the player wouldn't know they were crossing between sections.	There are no visible borders between terrain sections when viewed in both scene and game view. The player crosses the border between two sections with no notable difference to standard gameplay. Passed.	Test 1f
1gi)	Test that GameObjects are created with all components required for a terrain mesh.	Default Noise, Default Terrain	All terrain sections have appropriate mesh components and so are rendered in the game scene.	All terrain objects show an active MeshRenderer and MeshCollider component, such they are visible and can be collided with. Passed.	Test 1gi
1gii)	Test that the terrain collider enables the player to walk on the ground.	Default Noise, Default Terrain	The collides with the surface of the terrain mesh and can walk around freely.	The player can walk in any direction. Their movement, as expected, is only influenced by the steepness of the	Test 1f, 1gii

				ground or any obstacles. Passed.	
1giii)	Test that on the mesh that the player is standing on, the terrain collider accurately reflects the rendered mesh.	Default Noise, Default Terrain	The terrain collider matches the rendered mesh. Meaning that the player doesn't appear to be standing above or within the rendered mesh.	The terrain's collider accurately reflects the rendered mesh. Such the terrain that the player collides with is equivalent to that which they see. Passed.	Test 1f, Test 1gii
2a)	Test that natural game objects are spawned at realistic terrain heights.	Default Noise, Default Terrain 0.361 (Limit)	Trees are not spawned in the ocean, nor on the peak of mountains.	Trees are only spawned on realistic land types given the tree prefab used. No trees exist in the ocean or on mountain tops, they begin to fade in as the terrain becomes grassier. Passed.	Test 2a, Test 2b
2b)	Test that natural game objects are spawned with a defined separation.	0.361 (Limit) 8 (Spawn Radius)	All trees are separated such that they may overlap leaves, but not branches with one another.	There are no visible trees that overlap branches with each other. Passed.	Test 2b
3a)	Test that an enemy moves randomly while the player remains undetected.	108 (FOV Angle), 10 (FOV Distance)	The enemy's patrol is random in direction and time. The system log displays the length of each direction patrol, and a difference is evident between each. There is no log to indicate that the	The enemy's movement consists of both random direction and duration. The log outputs further support that the patrolling is random, and at no point indicates that the player has been	Test 3a

			player has been detected.	detected by the enemy. Passed.	
3bi)	Test that an enemy detects a player once the player is within the FOV of enemy.	108 (FOV Angle), 20 (FOV Distance)	The enemy detects the player when they are within both the FOV distance and angle. The system log indicates that the player has been detected.	The player stands within the FOV range and angle of the enemy. The enemy turns slightly to focus on the player. The log message appears to notify that the player has been detected. Passed.	Test 3bi
3bii)	Test that the enemy doesn't detect the player while the player is within the FOV radius of an enemy, but the enemy is facing away from the player.	108 (FOV Angle), 20 (FOV Distance)	The enemy doesn't detect the player while the enemy is facing away, despite the player being within the FOV radius of the enemy. There is no log to indicate that the player has been detected.	The player stands within the FOV radius on the enemy, but isn't detected, as the enemy is facing away. No log appears to suggest that the player has been detected. Passed.	Test 3bii
3biii)	Test that the enemy doesn't detect the player while the player is within the FOV angle of an enemy, but not within the minimum distance.	108 (FOV Angle), 10 (FOV Distance)	The enemy doesn't detect the player whilst the player is further away than the minimum detection distance, but within the FOV angle of the enemy. There is no log to indicate that the player has been detected.	The player stands with the enemy facing towards them, but further away than the FOV radius. The player remains undetected and there is no notification in the log that the player has been detected. Passed.	Test 3biii
3c)	Test that, while the player remains detected, the enemy moves towards the	108 (FOV Angle), 20	The system log outputs that the player is detected. The player's	The player is detected by the enemy, clarified by the message in the	Test 3c

	player's current position.	(FOV Distance)	movement diverts the course of the enemy for the duration of the player's detection.	log. The enemy moves towards the player and changes the direction of its movement based on the player's current position. Passed.	
3e)	Test that opaque objects in the game world can obscure an enemy's view of the player.	108 (FOV Angle), 20 (FOV Distance)	The player is within the FOV of an enemy and the system logs that the player is detected. The player's movement behind a tree causes the player to become undetected. The system-log no longer outputs that the player is detected.	The player is detected by the enemy, with an accompanying log message. The player's moves behind a tree and becomes undetected. The enemy begins patrolling and the system log messages cease. Passed.	Test 3c
4a)	Test that the player can orbit the camera around themselves within the defined lower clamp angle.	5 (Camera Distance), 70 (Clamp Angle)	The player can orbit the camera both left and right and up and won. The player can return the camera to directly behind them. A log message appears to indicate that the player has repositioned the camera.	The player can flexibly move their camera spherically around their position. The player presses the 'Z' key and the camera immediately returns behind them. The system log outputs that the player focused the camera. Passed.	Test 4a
4b)	Test that the camera will manoeuvre around an object that would obscure its view of the player.	5 (Camera Distance)	The camera is in its standard position until the player moves such that there is a tree between them. The	The player rotates the camera to a position behind a tree. The camera moves forward until it passes the tree. Once the camera is rotated past the	Test 4b

			camera moves in front of tree.	tree, it returns to its standard radius. Passed.	
4c)	Test that the camera remains a fixed distance away from the player.	5 (Camera Distance)	The system log outputs the distance of the camera from the player at multiple positions to be the same.	Over all positions sampled, the distance was found to be the same. Passed.	Test 4b
5ai)	Test that an object is spawned on the map that when collected will reveal the player's location in their map display.		The player collects the object and when subsequently viewing their inventory, a marker is displayed to show their current position.	The player is able to locate the 'Player Finder' object and having collected it, a marker appears on the player's map display in a position that accurately represents their own. Passed.	Test 5a
5aii)	Test that the player is notified when they collect the 'self-locating' object, and that the object is removed from the game scene		The player collects the object and a notification describing the function of the object is displayed. The object becomes inactive and is no longer visible in the position where it was collected.	Once collected, the object is no longer visible in the game view. The hierarchy shows that the object has become inactive. Passed.	Test 5a
5bi)	Test that an object is spawned that when collected will reveal the position of all enemies through the player's map display.		The player collects the object and when subsequently viewing their inventory, icons are shown on the map display, marking the position of all	The player is able to locate an 'Enemy Finder' object. On collection, the player's minimap and map display update to detail the positions of all	Test 5b

			enemies in the game.	active enemies. Passed.	
5bii)	Test that the player is notified when they collect the 'enemy finding' object, and that the object is removed from the game scene.		The player collects the object and a notification describing the function of the object is displayed. The object becomes inactive and is no longer visible in the position where it was collected.	Once collected, the object is no longer visible in the game view. The hierarchy shows that the object has become inactive. Passed.	Test 5b
5ci)	Test that an object is spawned that when collected will reveal a compass on the player's minimap.		The player collects the object and their minimap changes to display a compass that displays the north direction in relation to the rotation of the camera.	The player is able to locate a 'Compass' object. Once collected, the player's minimap now displays the letter 'N', which rotates as the player's camera moves such that it always points to north. Passed.	Test 5c
5cii)	Test that the player is notified when they collect the 'compass' object, and that the object is removed from the game scene.		The player collects the object and a notification describing the function of the object is displayed. The object becomes inactive and is no longer visible in the position where it was collected.	Once collected, the object is no longer visible in the game view. The hierarchy shows that the object has become inactive. Passed.	Test 5c
5di)	Test that an object is spawned that when collected will suggest		The player collects the object and when subsequently viewing their	The player is able to locate a 'Pathfinder' object. Having collected this, the	Test 5di

	a path to the goal collectable.		inventory, a path is drawn on the map display from their current position to the position of the goal collectable.	map display now marks a path from the player's position to the goal collectable. Passed.	
5dii)	Test that the player is notified when they collect the 'path-finding' object, and that the object is removed from the game scene.		The player collects the object and a notification describing the function of the object is displayed. The object becomes inactive and is no longer visible in the position where it was collected	Once collected, the object is no longer visible in the game view. The hierarchy shows that the object has become inactive. Passed.	Test 5di, 5dii
5diii)	Test that the suggested path avoids steep ascents where possible.	30 (Enemy Coefficient), 30 (Cost Coefficient), 1.1 (Height Coefficient), 0.5 (Heuristic Coefficient)	The path should avoid travelling up the line of greatest slope of parts of the terrain.	The path manoeuvres around brown areas of the map, suggesting that the path is avoiding steep ascents. Passed.	Test 5di, 5dii
5div)	Test that while the player has the 'enemy-finding' object, the suggested path avoids entering close-proximity of enemies.	30 (Enemy Coefficient), 30 (Cost Coefficient), 1.1 (Height Coefficient), 0.5 (Heuristic Coefficient)	The path avoids entering the spherical radius of enemies marked on the map display.	The path was unable to be generated. Error: Index out of range. Failed. The path was generated and visibly avoids entering the radius of enemies.	Test 5dii
5dv)	Test that while the player doesn't have the enemy finding	30 (Enemy Coefficient), 30 (Cost	The path takes the best route regarding the steepness of the	The path avoids travelling up the mountain, and	Test 5di

	object, the suggested path is impartial to the location of enemies.	Coefficient), 1.1 (Height Coefficient), 0.5 (Heuristic Coefficient)	terrain. The path may or may not pass through enemies.	happens to pass through the radius of two enemies, but appears unaffected by their presence. Passed.	
6ai)	Test that the player's health is displayed as part of a HUD while the game is active.		The player's health is shown on screen in both active and paused mode. The log indicates the damage taken when the player is hit by an enemy, the player's health display adjusts by the same amount.	The player's health is always displayed in the top-left corner of the screen during gameplay. The player is attacked by an enemy causing the log to report a loss of 5 health. This change then appears to the player's displayed health. Passed.	Test 6
6aii)	Test that the game ends should the player's health fall to 0.		The player receives a hit from an enemy causing their health to fall to 0. An appropriate UI pop-up is displayed and the game ends.	The player's health falls to 0 as they are hit by an enemy. The game-over message appears on screen and the game ends. Passed.	Test 6
6aiii)	Test that a minimap showing a birds-eye of nearby terrain is displayed in the player's HUD while the game is in an active state.		The player's minimap is displayed on screen, and updates as the player moves and rotates themselves or the camera.	The minimap is displayed in the top-right corner of the player's screen. The minimap rotates with the camera. Passed.	Test 6, Tests 3, Tests 4, Tests 5
6aiv)	Test that the player's in game minimap isn't displayed while the game is paused in the inventory menu.		While the player is accessing the inventory menu, the in-game minimap is no longer displayed,	The minimap doesn't appear whilst the player is viewing the	Test 6, Tests 5

			the map display is shown instead.	inventory menu. Passed.	
6ci)	Test that the player is able to bring up an inventory menu displaying the entire map.		The player presses the pause key to display the inventory menu. The log indicates that the inventory menu has been entered, and an image showing the entire map from above is displayed.	The player presses 'P' to access the inventory and the map display appears, showing a top-down view of the entire map. The log indicates that the player has entered the inventory menu. Passed.	Test 6, Tests 5
6cii)	Test that the game is paused when viewing the inventory menu.		The player currently being attacked by an enemy presses the pause key. Both the enemy and the player are frozen in place.	The pause key freezes the game. Passed.	Test 6
8ai)	Test that the identical terrain can be regenerated.	Default Noise, Default Terrain Default Noise, Default Terrain	By entering the same noise and terrain data, the resulting terrain is identical to the previous.	The two terrains generated using the same input data are identical. Passed.	Test 8

ERRONEOUS DATA

Such is the nature of my project that there are very few opportunities where it would be possible for the user to input some form of erroneous data. This is because the user inputs very little data throughout the game, that which they input is controlled by UI elements that always return data of a given type. In game, should the user enter any key that I haven't mapped to a function, then nothing will happen. This is a built-in feature of Unity that prevents me from having to validate all of the user's input. Therefore, it is essentially impossible for erroneous data to occur as a result of user input. Such, all causes would occur from within the system, and should be identified by my test-plan, as it tests all parts of the system.

TESTING EVIDENCE

In this section I will provide evidence for each of the tests that I planned and conducted. For each test I will provide appropriate visual evidence, using images or a video. In each case, where relevant, I will include in the evidence the test data's entry.

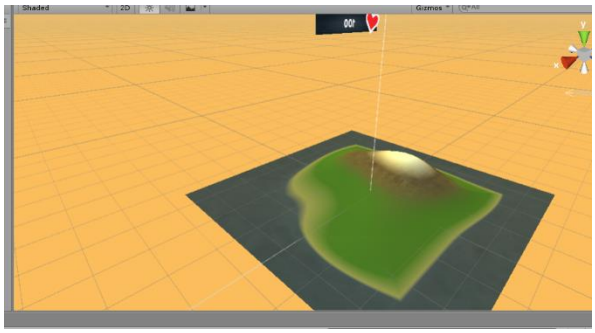
Objective 1

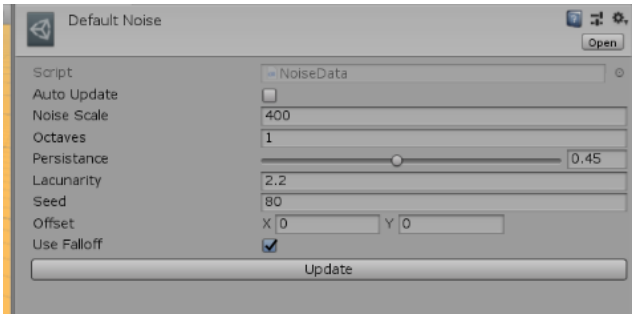
Test 1a)

As shown, I have input an octave count of 7 in the inspector. Along with this, some standard noise data is input. The logger then shows that 7 is in fact the octave count being used. Furthermore, during the use of each octave the logger outputs which octave is currently being iterated through.

Test 1aii)

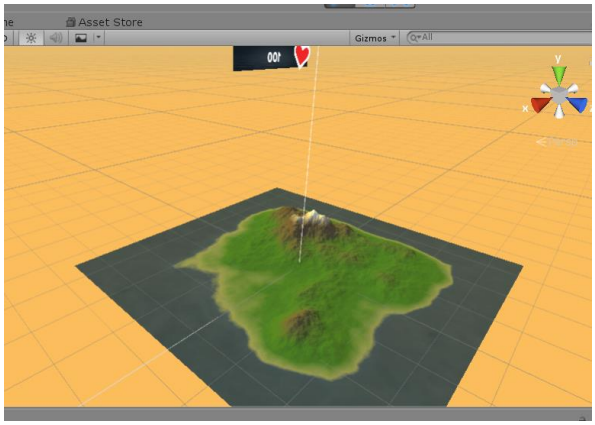
Using 1 octave:

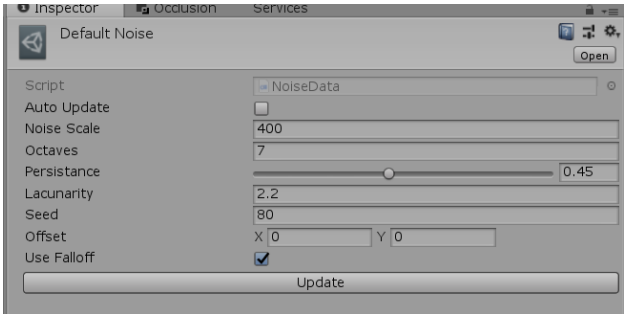




These images show the resulting terrain when I have input 1 as the number of octaves to be iterated through in generation. I have disabled object-placement to make the terrain more visible. From the evidence we can see that the resulting terrain is extremely smooth and continuous.

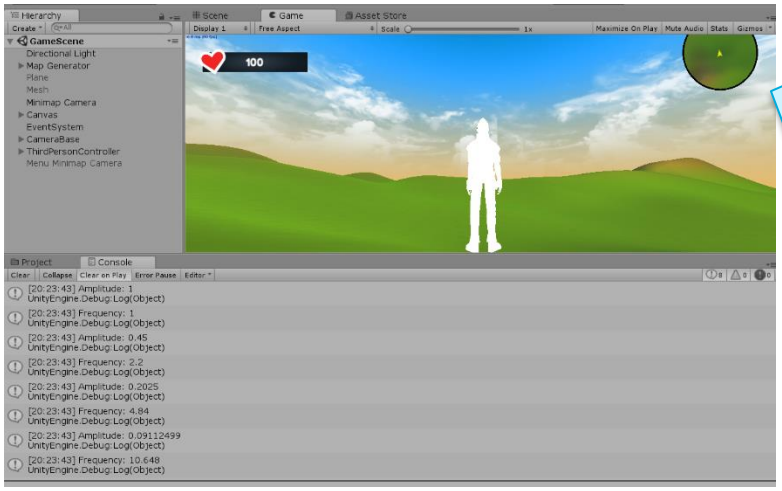
Using 7 octaves:





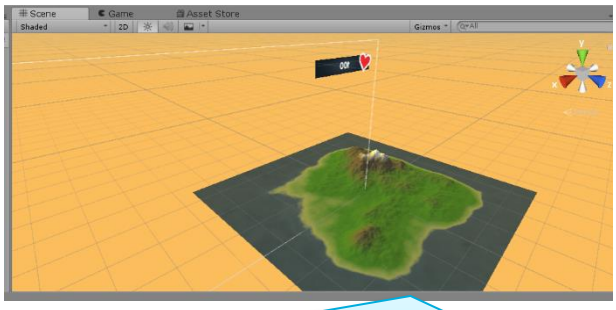
These images show the resulting terrain when I have input 7 as the number of octaves to be iterated through in generation. In comparison to the previous images the terrain is much more detailed and angular. As shown, all other input noise data is the same, so we must conclude that this due to the program overlaying Perlin Noise multiple times.

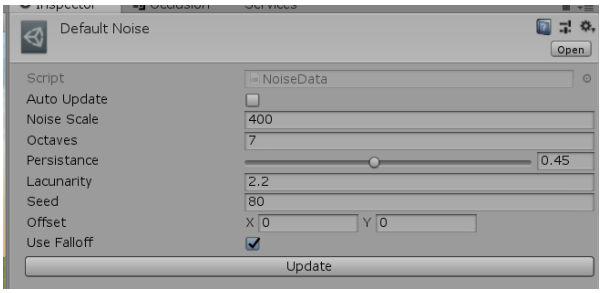
Test 1b)



From this evidence we can see from the log that amplitude and lacunarity are changing between octaves. As both begin at 1, in the second octave, their values become equal to persistence and lacunarity respectively. From here calculations verify that both are changing by a factor of their modifier each time.

Test 1c)

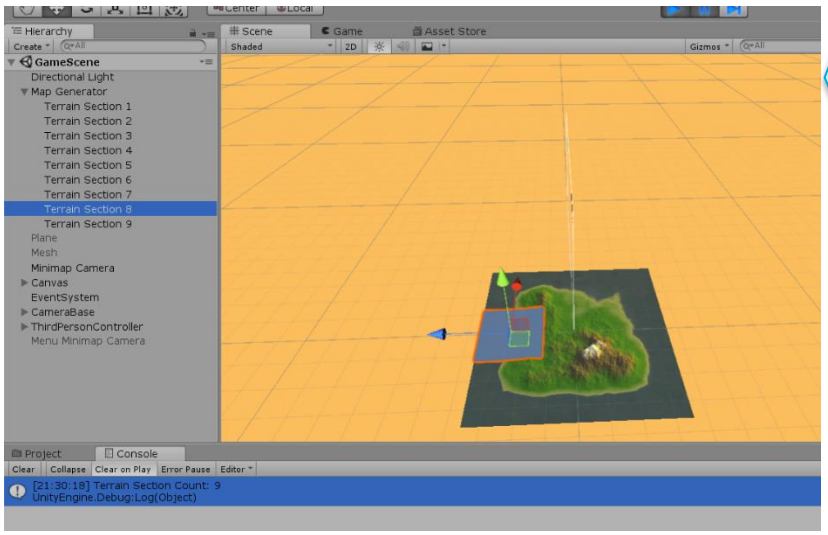




I have used the above images to show a visualisation of the terrain from different viewpoints. I have included a view of the main game camera (upper left), the player's map display (upper right) to display a bird's eye view, and a view from the scene view camera (lower left). From these images I think that it is evident that the terrain is varied and diverse. It includes large grassland areas, along with mountains and beach sections as well.

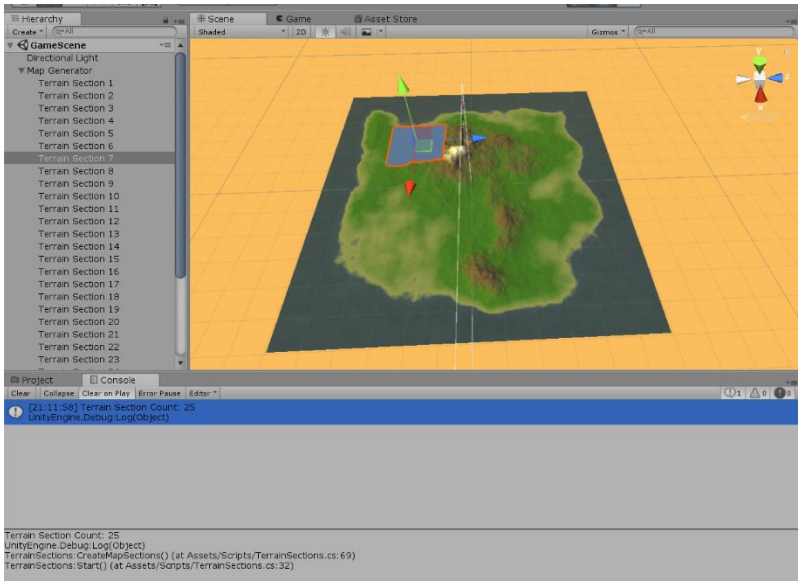
Test 1ei)

Using a 3x3 map:



This image shows the terrain created when a map size of 3x3 is used. As expected, there a 9 terrain sections in the scene, shown by the number of sections in the scene hierarchy, and the message shown by the log. I have also selected a terrain section to highlight its size relevant to the map, as it helps to demonstrate that the map is 3x3.

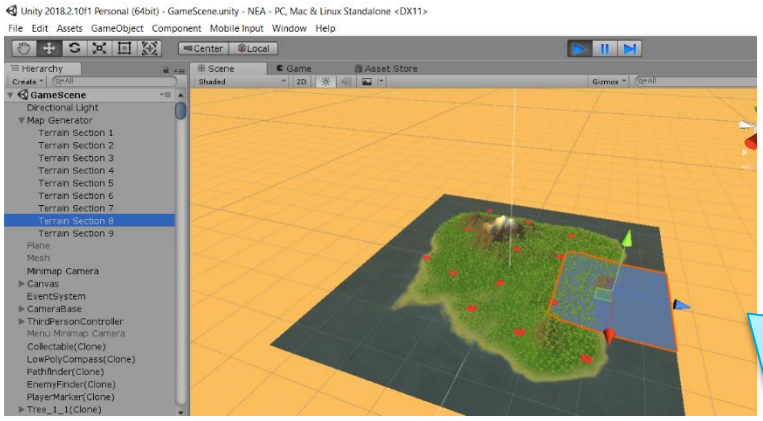
Using a 5x5 map:



This image shows the map having been created with input map dimensions as 5x5. As shown by the test data images, all other terrain values are the shared between the two samples. Once again, I have highlighted a terrain section to illustrate the dimension sizes. Furthermore, shown in the scene hierarchy are 25 terrain sections and displayed in the Console log is the message that 25 sections have been generated.

Test 1eii and 2a)

Using a 3x3 map:

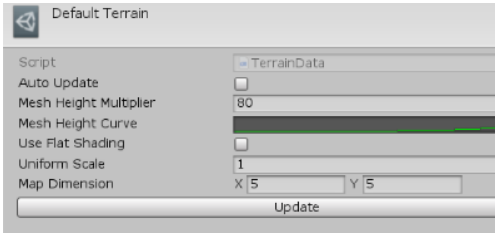


Test1eii)

This image shows that all object placement is appropriate scaled to the map size. As both enemies and trees can be found near the border. It is evident that the map is 3x3 from the terrain sections shown in the inspector, along with the size of a highlighted section in comparison to the entire map.

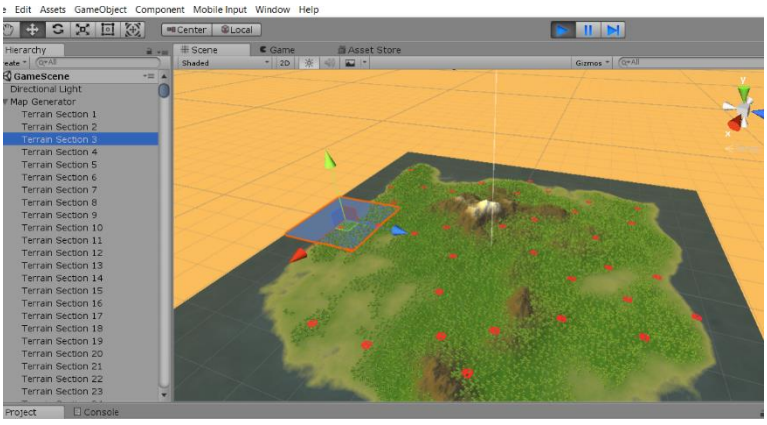
Test2a)

As visible from the image, trees are only spawning on grassy or light mountainous areas. No trees are spawned on the top of mountains, on beaches or in the sea. This is because the program is sampling the height of the terrain before choosing to spawn an object.



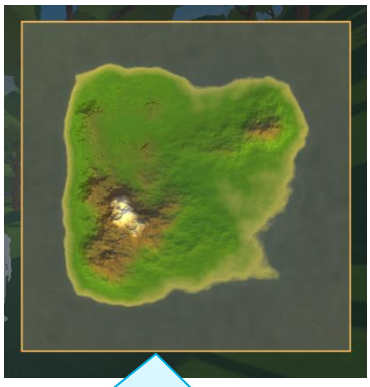
Using a 5x5 map:

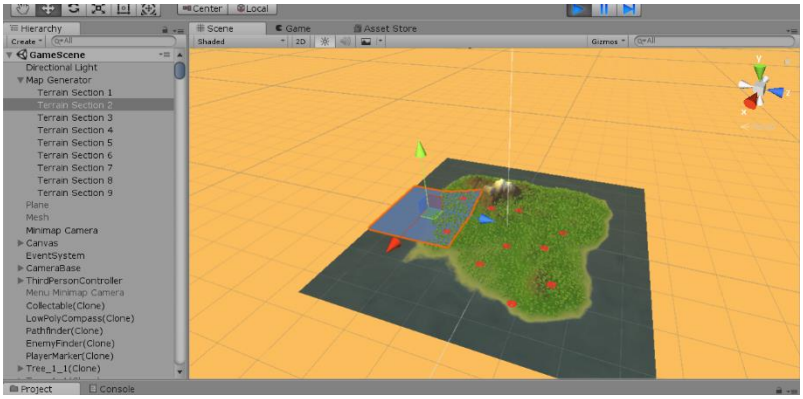
When using a 5x5 map, we can see that object placement has scaled appropriately. We can justify that the map is 5x5 from the scene hierarchy, and the proportional size of the highlighted terrain section. I have highlighted section 3 because it is a section that wouldn't be created in a 3x3. As we can see, it contains both enemies and trees, demonstrating that object placement is flexible with different map dimensions.



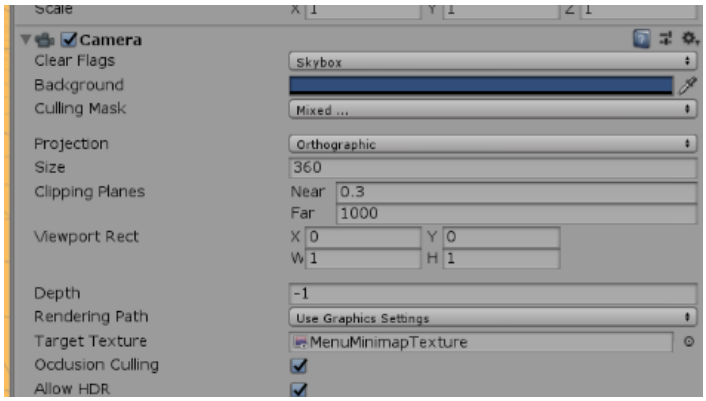
Test 1eiii)

Using a 3x3 map:

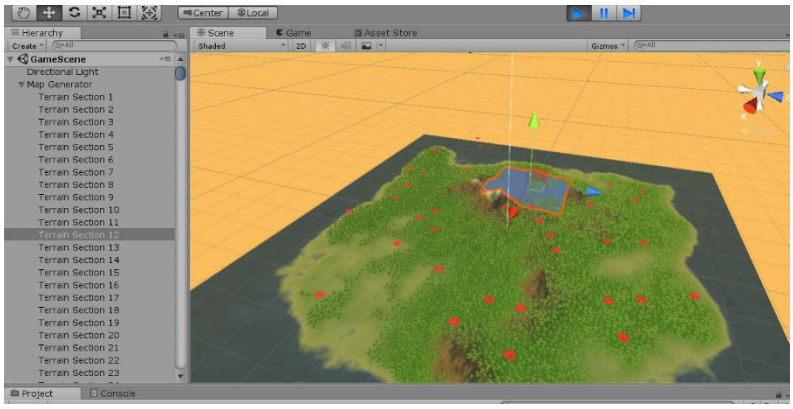




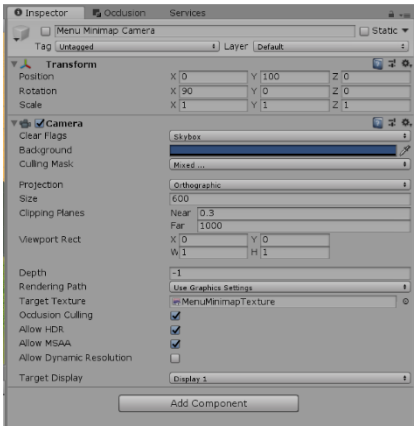
These images show the player's map display, giving a bird's eye view, compared with the scene view of the terrain. It is evident that both images show the same map. We can also see that the player's map display is scaled appropriately to fit within the 360 square-units UI image. The scene-view image also illustrates that the map is 3x3.



Using a 5x5 map:



We can see that the image has scaled appropriately to fit within the border (gold) of the map display. This means that the camera must have positioned itself accordingly. The accompanying scene view image illustrates that the map is 5x5. We can also see that the size attribute of the map-display camera has changed to 600.



Test 1f)

Video available from: https://www.youtube.com/watch?v=ICmxoMKvZlM&feature=youtu.be

In this video we can see the player – the plain white textured character – standing on 'Terrain Section 5' by the border of 'Terrain Section 6'. The video illustrates the line at which the sections meet, and there is no visible seam there in either scene or game view.

Test 1gi)

Video available from: https://www.youtube.com/watch?v=32XP9Wk3YOo

Here I have shown the components of each terrain collider on the map by scrolling through them in the hierarchy. It is evident that each terrain section has MeshRenderer and MeshCollider components that are active.

Test 1gii)

Video available from: https://www.youtube.com/watch?v=XGRGcwnnrcg

This video shows the player running around the map. From the player's camera we can see the rendered terrain exactly matches the mesh that he is colliding with.

Objective 2

Test 2b)

Video available from: https://www.youtube.com/watch?v=ERzfu_XTlpo

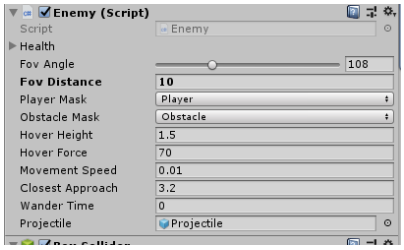
In this video we can see the player moving past quite a large sample of trees. In the video, no tree appears to have branches that cross those of another tree. This case is also seen in all of the other test videos that feature trees.

Objective 3

Test 3a)

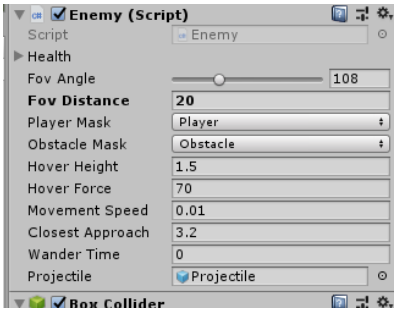
Video available from: https://www.youtube.com/watch?v=6YnEYjsKMn8

I have reduced the FOV radius from the standard value of 20 to 10, such that it is easier to observe the behaviour of the enemy without being detected. The input values are shown here as an image. In the video we can see the log updating each time the enemy conducts a new wander, along with the duration and angle of this movement. There is no message in the log to suggest that the player has been detected.



Test 3bi)

Video available from: https://www.youtube.com/watch?v=tK2hoJuIG_4



For this test I have returned the FOV distance to the standard value of 20. In the video we can see the player move within the FOV of the enemy. This causes the enemy to face the player and begin firing. Along with this, the log outputs, every 0.2 seconds, that the player is currently detected.

The video also features transform values of both the enemy and player at a point whilst the player was detected, from this we can easily calculate whether the player is within the detection radius of the enemy. We can see that the player safely falls within the FOV radius and should be detected.

Distance:

Player: X : -6.16 Y : 10.17 Z : -0.07

Enemy: X : -3.18 Y : 10.90 Z : 0.86

$$\boldsymbol{Distance = \sqrt{(-6.16 + 3.18)^2 + (10.17 - 10.90)^2 + (-0.07 - 0.86)^2}}$$

$$\boldsymbol{Distance = 3.21\ units} \qquad \boldsymbol{3.21 < 20}$$

Test 3bii)

Video available from: https://www.youtube.com/watch?v=Yno6jnmdl7U

In this video, the player moves within a distance that we presume is less than the detection radius of the enemy. But the player isn't detected, as shown by the absence of firing and log messages. Thus, we can conclude that this is because the player isn't within the FOV angle of the enemy's forward vector. We can verify these values with some calculations.

Distance:

Player: X : 2.76 Y : 9.18 Z : 1.12

Enemy: X : 1.43 Y : 8.77 Z : 4.08

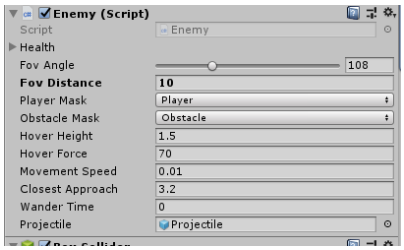
$$\boldsymbol{Distance} = \sqrt{(\mathbf{2.76 - 1.43})^2 + (\mathbf{9.18 - 8.77})^2 + (\mathbf{1.12 - 4.08})^2}$$

$$\boldsymbol{Distance} = \mathbf{3.27}\ \boldsymbol{units} \qquad \mathbf{3.27 < 20}$$

Test 3biii)

Video available from: https://www.youtube.com/watch?v=BsAEXynUFig

For this test I have reduced the FOV radius down to 10 once again. We can see from the video that the player is clearly within the FOV angle of the enemy, as they are almost facing directly towards each other. However, there is no log to indicate that the player has been detected, suggesting that the player isn't within the FOV radius, once again we can check whether this is the case using transform values shown in the video. As shown by the calculation, the player shouldn't be detected as they don't meet the distance condition.



Distance:

Player: X : 10.67 Y : 10.90 Z : -16.44

Enemy: X : -0.38 Y : 12.03 Z : -11.61

$$\boldsymbol{Distance} = \sqrt{(\mathbf{10.67 + 0.38})^2 + (\mathbf{10.90 - 12.03})^2 + (\mathbf{-16.44 + 11.61})^2}$$

$$\boldsymbol{Distance} = \mathbf{12.11}\ \boldsymbol{units} \qquad \mathbf{12.11 > 10}$$

Test 3c)

Video available from: https://www.youtube.com/watch?v=AXFowiY835A

The video begins by showing the player in a detected state. We can tell this from the fire of the enemy, and the log messages in the bottom left corner of the screen. It is clear that the enemy moves towards the player's current position, as the enemy begins reasonable far from the player, and reaches quite closer. Furthermore, we can see that as the player moves side to side, the enemy changes its movement to reflect this. The player then moves behind a tree, causing them to become undetected this is shown by the movement away by the enemy and the log message showing that the player is 'Now Undetected'.

Objective 4

Test 4a)

Video available from: https://www.youtube.com/watch?v=A58Gm9iL5V4

In this video we can see the player moving around the map and rotating the camera around themselves. The movement is very smooth as it is updated each frame and the time between frames is accounted for. The player then presses the 'Z' key which focuses the camera directly behind them. The player then repeats this 3 more times, and a message in the log appears each time.

Test 4b)

Video available from: https://www.youtube.com/watch?v=MsUcfZoXGGI

In this video we can see that each frame, the camera is outputting its distance from the player, being 5 units. When the player clamps the camera to the floor, or rotates past a tree, we can see the camera move closer the player. This is reflected in the log messages that show a clear decrease, however once an obstruction is cleared, the distance returns to 5. At the end of the video, I have highlighted that the standard distance should be 5, proving the success of this test.

Objective 5

Test 5a)

Video available from: https://www.youtube.com/watch?v=7SSdj_y8hl4

The player begins by running into the 'Player Marker' collectable. They are shown a pop-up with relevant information, and the game pauses during this period. When subsequently viewing their map display, the player now has an oscillating marker to show their position on the map. In the active game, it is clear that the collectable has disappeared from view. This is verified when the scene hierarchy is checked, and the item is shown to be inactive. I have also used the scene view to verify the validity of the marker, it can be seen that the ThirdPersonController's (Player) position is the same as that of the marker.

Test 5b)

Video available from: https://www.youtube.com/watch?v=rX4APSLhg2Q

The player begins by running into the 'Enemy Finder' collectable. They are shown a pop-up with relevant information, and the game pauses during this period. The player then approaches reasonably near to an enemy, in this time it is clear that the collectable has disappeared from view. An icon representing the enemy can now be seen on the player's in-game minimap. The player next views the inventory menu, where the map display now marks the positions of all enemies on the map. When checking the scene hierarchy, the 'Enemy Finder' object is shown to be inactive.

Test 5c)

Video available from: https://www.youtube.com/watch?v=I92ydV2wG-0

The player collects the 'Compass' item and their in-game minimap updates to show the marker. The player is seen to be facing the mountain, the compass points in the opposite direction. When checking the map display, we can see that north is indeed where the marker was pointing. We can also see the compass rotating in reverse to the camera such to always point towards the same direction. Once again, the scene hierarchy shows that the item has become inactive.

Test 5di)

Video available from: https://www.youtube.com/watch?v=8r6d9QnoJRE

Having collected the 'Pathfinder' item, the player accesses their inventory menu. A cyan-coloured path is drawn from the player's position to the goal collectable. The video then exits to scene view. From here we can see that the goal is indeed where the path suggests travelling to. We can also see from the position of enemies, that the path isn't influenced by them as it travels straight through their FOVs. This is because the player hasn't acquired the enemy finder object.

Test 5dii)

Video available from: https://www.youtube.com/watch?v=8r6d9QnoJRE

This video shows the program failing to meet the objective 5. This is because, when the player tries to view the inventory, an 'Index Out-of-Range' exception is thrown by the graph. This suggested that my calculation of graph index was flawed in some way.

The Fix:

I found that when indexing the graph, it was possible for the enemy's position plus their FOV to exceed the size of the array. This meant that I was trying to reference a position that didn't exist. To fix this I introduced selection that would only influence the graph if the index was valid.

Video available from:
https://www.youtube.com/watch?v=IYBYP9ri8oY

The test was now successful. The path drawn avoids both entering the proximity of enemies and ascending steep terrain. It can be seen in the scene hierarchy that the path is created to the correct location, and that the pathfinder object has become inactive. I have included a variety of images to showcase the pathfinding in different scenarios and maps.

Objective 6

Test 6)

Video available from:
https://www.youtube.com/watch?v=buIJ5bgG714

In this video we can see the player's health in the top-left hand corner of the screen. We can also see the in-game minimap shown on the other side of the screen. When the player is hit by the enemy's projectile, the log indicates that the player has lost 5 health. This is reflected in the change to the player's health meter. When the player accesses the inventory menu, the map display appears and the in-game minimap is no longer visible. We can also see that during this time, the player is taking no

damage from the enemy, nor is their camera moving, the game is paused. When the enemy reduces the player's health to 0 a game-over pop-up is displayed and the application closes.

Objective 8

Test 8)

Video available from: https://www.youtube.com/watch?v=bwYLkllvR68

Here we can see the default noise and terrain input data being used to generate two maps consecutively. We can see that both terrains are identical, however the object placement and enemies are not. This means that I have not fully met this extension objective.

Evaluation

EVALUATION AGAINST OBJECTIVES

In this section I will try to determine the success and failures of my project by comparing its performance against the system objectives that I defined in the analysis section. By doing this, I will be able to judge whether I met my initial goals, exceeded them, or failed to reach them.

Objective 1: PROCEDURALLY GENERATE RANDOM, TRAVERSABLE AND CHARACTERISTIC TERRAIN USING A 3D NOISE FUNCTION WITH APPLIED VARIABLES.

Objective 1a): OVERLAY NOISE IN MULTIPLE OCTAVES TO INCREASE THE DETAIL OF THE TERRAIN.

The project meets this objective. The program lays multiple octaves of Perlin Noise on top of each other using a 2D array. The entire array is then standardised between 0 and 1 by using the highest recorded value as a maximum. The number of octaves used is easily altered and the resulting effect on the terrain is quite evident, with a greater number of octaves decreasing the unrealistic smoothness and lack of detail in the terrain.

Objective 1b): USE LACUNARITY AND PERSISTENCE TO ALTER THE FREQUENCY AND AMPLITUDE BETWEEN OCTAVES.

The project meets this objective. The amplitude and frequency are multiplied by persistence and lacunarity respectively at the end of each octave. Higher values for persistence give more violent and random terrain. Higher values of lacunarity increase the spread of change between octaves. These values were extremely important in giving the terrain a more realistic feel.

Objective 1c): USE DIFFERENT STYLES OF TERRAIN TO ENSURE THAT THE 'WORLD' IS SUITABLY VARIED.

The project meets this objective. The terrain can be found as water, sand, grass, light rock, dark rock and snow. This gives the terrain a great range of variety in colour and texture. To achieve this, the terrain height at each point is sampled and an appropriate change is made to the texture shader to reflect the height. This helps provide a realistic feel, as grass and water are often found in lower regions, and snow and rock found in higher regions. The shader uses tri-planar mapping to ensure that the texture isn't stretched when applied to a larger surface area.

Objective 1d): CREATE THE TERRAIN AS AN ISLAND USING THE INFLUENCE OF A FALLOFF MAP.

The project meets this objective. The program creates a 2D array of falloff values to be subtracted from the initial height-map. As the falloff map holds a value of 1 at its edges, all noise-map values around the outside of the terrain become 0 or less. All of these sub-zero values can be set to 0 and the terrain then re-standardised between 0 and 1. By defining a sea-level just above 0, all of the outer noise values become water and are coloured with a blue-texture. The multiple options for falloff equation also allow for increased variety in the rate of falloff.

Objective 1e): Vary the size of the map that can be created by joining terrain areas together.

The project meets this objective. A large range of map sizes can chosen for world generation, which is beneficial for improving game variety. The program records the map-dimension as a Vector2 variable and uses it in any calculations that need to scale between multiple map sizes, this is what allows the program to be so accommodating. Such, falloff-maps can be used on each map size, and object placement will fill the landmass of all terrain sizes used. The disadvantage, of course, that comes with increasing map size is the increase in expense on the CPU and GPU. This increase is additionally a square-function as it is based on map-area, meaning that increasing map size can become exceedingly difficult at a point.

Objective 1f): Using the noise values as vertices, create a mesh with an accurate terrain collider.

The project meets this objective. The player can walk on all terrain areas, and the rendered terrain matches the shape that the player collides with. This is achieved by multiplying each noise value by a sizing constant, then by transferring each noise value into a vertex array. Accompanying this, a triangle array that holds vertex indices and a UV array must be created. With these three arrays, the Unity engine can create a mesh and place it in the scene.

Objective 1g): Join terrain areas together such that there is no visible seam.

The project meets this objective. The program makes sure that each noise-map is created before standardisation, this means that an overall maximum height can be found to be used for standardisation of all sections. The program then gives the noise-map array for each terrain section a border. In this border, the height of each vertex in the neighbouring terrain can be stored, this means that when lighting the mesh, a triangle can be created using a vertex from a different terrain section. Despite the triangle not itself being rendered, its presence is paramount as it will determine the normal of all of its vertices and should any of these vertices exist in the mesh, then that vertex will be rendered and used in active triangles. Such, the border corrects the lighting of all triangles along the edge of mesh as if the map was one large terrain section.

Objective 2: Randomly spawn natural biomes onto the terrain in suitable and realistic locations.

Objective 2a): Use terrain height to find appropriate locations for natural game objects.

The project meets this objective. In a very similar way to the application of the terrain shader, the height of the terrain is sampled to determine whether object placement is possible. For tree placement, any points deemed to be in an unsuitable height location don't have a tree instantiated at their position in world space. It was important in many sections of the program to convert from the array space to the world space, this was the case in object spawning. This is because array indexing begins at 0, but the map existed equally in both positive and negative parts of the x and z axes.

Objective 2b): Spawn natural game objects with characteristic separation or frequency.

The project meets this objective. For spawning trees, the program uses the Poisson Disc Sampling algorithm to fill grassy areas with trees efficiently and ensure that they are appropriate spaced and randomly distributed. I believe this distribution to be characteristic of trees as they are often found in large groups, but never too close to diminish their own growth.

Objective 3: Create a detection system for the enemies such that they can locate and approach the player.

Objective 3a): While the player is undetected, the enemy should patrol in random directions

The project meets this objective. The enemies store the condition of the player's detection as a Boolean field in their class, this is because detection is only applicable to each enemy as a separate attribute, as if one enemy detects the player, it doesn't mean that all enemies should consider the player detected. Such, while an enemy's detected value remains false, it generates a random angle and time. The enemy then moves at that angle for the given time before repeating again. This makes the enemy's patrolling unpredictable for the player, meaning that it would be dangerous to enter their field of view at any time, should they turn and spot the player.

Objective 3b): Use a visibility algorithm such that the player can only be detected by an enemy when they're within a certain radius of the enemy, and within the enemy's line of sight.

The project meets this objective. The enemies use Pythagoras' theorem with their world position and the player's world position to determine whether their separation is less than the enemy's FOV radius field. To determine the angle, an enemy finds the angle between their forward vector and the vector to the player using a vector dot product.

Objective 3d): If the players the enemy or is detected, the enemy will move towards the player as long as the player remains detected.

The project meets this objective. By storing a reference to the player GameObject, the enemies are aware of the player's current position. They can then, in Monobehaviour's Update() method, change their path towards that point.

Objective 3e): Objects in the world will obscure the enemies' view of the player, enabling the player to become undetected.

The project meets this objective. The enemies make use of Unity' Raycasting feature to determine whether the view of the player is obscured in any way. This is done by firing a virtual ray from the enemy's world position to the player's world position. Should the ray strike any object before reaching the player, it can be concluded that the enemy's view of the player is obscured, and the detected field can be set to false;

Objective 4: Implement an interactive 3rd person camera system.

Objective 4a): The player should be the focus of the camera during standard gameplay, with the player having the ability to orbit the camera around themselves at any time.

The project meets this objective. As a child of the player's GameObject, there is a Camera Focus object positioned around the player's upper-body. The main camera focuses on this object at all times, meaning that the player remains the target. The camera's script takes mouse or controller input in the Update() method and converts the value to rotation of the camera about the focus object in the input direction.

Objective 4b): The camera should follow the path it is led by the player, but automatically move around obstacles that would obscure the view of the player.

The project meets this objective. Like the enemies, the camera uses the Raycasting feature to determine if any objects are obscuring the view of the player. The great benefit about Raycasting is that it provides information about the collider that was hit. As the raycast records the position of the hit, the camera can move itself along the vector to the player, but position itself just in front of the object that was previously blocking its view of the player.

Objective 4c): The camera should remain a fixed distance away from the player unless it is forced to manoeuvre.

The project meets this objective. The camera's desired distance from the player is stored as a floating-point number. The camera will use this distance and its current vector to the player to find its desired point. The camera will then in the Update() method move towards that desired position. Such, the camera will always try to remain this input distance away from the player. When experimenting with the implementation of saving values between games, I added a settings menu with a slider that allows the player to adjust the standard camera distance from the player.

Objective 5: Spawn a range of collectable items in the world to assist the player in traversal or combat.

Objective 5a): Spawn an object that will reveal the player's location in relation to the entire map.

The project meets this objective. When the player collects the 'Player Finder' object, the player is displayed a UI panel that is passed appropriate information from the Manager class. In addition, a marker UI GameObject becomes active and appears over the map display. The marker is passed the player's position in world space and the map dimensions. With this the marker can calculate the player's position in proportion to the entire map, and thus position itself according on the player's map display. My end user had suggested that it look good if the marker were to oscillate up and down. To implement this, I made use of Unity's coroutine system. This is because the oscillation would have to occur when the game was paused, and when this is the case, the time-scale is 0, and so Update() methods don't occur. Such I made a coroutine that uses the WaitForSecondsRealtime function to ignore the game timescale.

Objective 5b): Spawn an object that will reveal the position of all enemies on the map.

The project meets this objective. On each enemy prefab there is a large icon positioned about 20 units above the enemy. This icon has no collider and isn't rendered by the main camera at any point. However, once the player collects the 'Enemy Finder' GameObject and are shown an appropriate message. The player's minimap and map display cameras have their culling masks updated using a bitwise shift operation.

Objective 5c): Spawn an object that will add a compass to the player's navigation system.

The project meets this objective. When the player collects the 'Compass' GameObject they are once again shown a message with relevant information. Then, a UI GameObject that marks the direction becomes active on the player's minimap. The north marker spins around the minimap based on the player's main camera rotation. To do this, the marker stores a centre point, being the middle of the player's in-game minimap. The x and y positions of the marker are then calculated by taking Sine and Cosine respectively of the player's main camera rotation around the y-axis. This causes the marker to rotate such that it always marks north in respect to the player's camera rotation.

Objective 5d): Spawn an object that will reveal a selected path to the goal on the player's map.

The project meets this objective. When the player collects the 'Pathfinder' their map display is updated whenever viewed to show a suggested path to the goal collectable. This of course makes this item very powerful once obtained. The pathfinding graph is created using all of the noise-values used in terrain generation. Each of the noise values are used to create an instance of the class Coordinate. I have built this class to provide pathfinding functionality, as each Coordinate acts as a node. Such each coordinate stores data to speed up the pathfinding process, such as whether the node has been seen before, its cost, its heuristic e.t.c. As a cost function for movement between nodes, the absolute Tangent value of the angle between the two nodes is calculated and multiplied by the cost coefficient. This punishes steep ascent and descent; however, descent is only 0.4 times as costly as ascent. Each node also stores its proximity to the closest enemy. The movement cost between nodes is incremented by the average proximity cost of the two movement nodes, multiplied by a coefficient. A proximity value only exists if a node is within the current FOV radius of an enemy and is calculated by linearly interpolating the separation of the node from the enemy over the enemy's FOV distance. This results in the path avoiding enemies where possible, however the path will only do so once the player has collected the 'Enemy Finder' object.

Objective 5e) Spawn an object that the must be collected for the player to win the game.

The project meets this objective. When the player collides with this rotating cube object, they are displayed a UI message using the standard template stating that they have won the game. Having acknowledged the message, the application will then stop.

Objective 6: Create a simplistic and informative UI to display game information.

Objective 6a): Create a player HUD to display frame relevant information during the game.

The project meets this objective. To display the player's health a have added to fill component to a heart icon, alongside this is the player's health number out of 100. To keep this information frame relevant, the player's health is output in the Update() method. The mini-map is created using an orthographic camera that follow the player from above and renders the terrain beneath it. This is then cropped using a circular layer mask. The mini-map is set to rotate with the player's camera, not the player themselves, such to calculate the mini-map's rotation, the difference between them must found. The player's rotation is used directly as the rotation of the player marker on the mini-map. When the player presses the 'P' key to pause the game, the Manager class performs a NOT operation on the active state of the minimap to allow it to flip when the game changes state.

Objective 6b): Develop an inventory menu where the player has a large display of the entire map

The project meets this objective. When the player pauses the game, on the right hand of the screen a bird's eye view of the entire map is shown on an UI image. This is done by creating a render texture from the output of a statically positioned camera above the map. The manager class adjusts the orthographic size of the camera as a multiple of the map dimensions such that the image is always suitably sized for the UI panel. This image will also update its culling mask to render enemy objects should the player find the appropriate collectable. Furthermore, this display is used to show the player's marker and draw the suggested path to the goal collectable. This makes this feature an integral part of the player navigation. Despite the player not having in inventory in the current game, I'm happy with the way the map display turned out.

Extension Objectives

I was unable to fully meet any of my extension objectives due to time restraints. This is mostly due to my initial objectives taking longer to implement than expected, or that the extension objectives proved more demanding than I thought or that they were perhaps unsuitable for the direction that the project developed in. I do however feel that my project extended on objectives that I did meet, and in many cases achieved tasks that could've constituted a major objective themselves.

Objective 3c): The player's activity in the proximity of the enemy, such as making a noise can influence the detection algorithm.

The project doesn't meet this objective. I feel as though the current game lacks enough different game objects to introduce such a feature, but perhaps it would be possible to implement a system that took the player's movement speed and compared it with the player's distance from the enemy to determine whether the player should be heard. I could also take the type of terrain that the player is moving on and assign each type a sound constant, meaning that running on sand may be quieter than running on rock.

Objective 9: Create an interactive and skill-based combat system

The project doesn't meet this objective. I discovered that creating a combat-system would require gaining knowledge from scratch about animation or require implementing a 3rd party system. I felt that my time would be better spent developing the fundamentals in the game, however I don't disagree that a combat system would be a great addition to the game, making it aesthetically more impressive and more enjoyable to play. The lack of combat system does benefit the game in the sense that enemies become more of threat, giving the player more incentive to avoid them and try and find the suggested path.

Objective 10: Allow players to revisit previous islands.

The project doesn't fully meet this objective. I didn't implement a feature where the player visits successive islands, although I'm sure that it's a feature that I would add in future development. I experimented with a settings menu that would allow the player to set some preference for fields such as camera distance, using Unity PlayerPrefs feature and I think that a similar approach could be taken to terrain values and object and enemy positions. However, all ADT variables would have to be converted from for storage in this way. I can to an extent recreate an island, as the terrain will be identical if the same noise and terrain values are input, however the objects differ between simulations due to the random nature of the Poisson Disc Sampling algorithm.

EFFECTIVENESS OF SOLUTION

The decision to use Unity for the development of my project provided me with both pros and cons. My unfamiliarity with the software did slow development during early and mid-stages of the project, or when it came to interact with a new feature, such as UI. However, the software is well-documented, and the community offers lots of information, so I was always able to overcome the issues that were presented. In places it became difficult to fully implement the OOP model, due to the preferred behavioural nature of Unity's programming.

Despite this, Unity offered so many features to the benefit of the project, that I am glad about my decision to use it. Quick testing was easy in Unity, as I could easily change the values of fields that I chose to display in the inspector. The included physics engine shouldn't be taken for granted, as it dealt with many of the repetitive calculations that I would have to implement myself, and I think that creating my own physics engine would be worthy of a project in itself. Furthermore, creating a UI in Unity and integrating it with scripts is very simple. I feel that my UI is quite visually impressive, and so I have considered that perhaps making it simpler and focusing on implementing the extension objectives would have been a better plan, given my time-restraints. Although, my project effectively meets all of my initial objectives, and so I'm not strongly regretful of my decisions. Perhaps the most important feature, is the 3D graphics that Unity offers, providing me with a medium to achieve my project's aims. The GameObject and component system made changing the appearance of the game extremely achievable in just a few lines of code. This meant that it was quite easy to experiment with lots of different variables, objects or textures in a short space of time. Furthermore, it prevented the project from being cluttered with code that only manages the aesthetics.

I thought that the terrain generation was an interesting challenge, as there were so many ways to approach it. I'm pleased with the results of my approach, and believe that Perlin Noise was ideal to use, as it offered scope for increasing variation and complexity, without overcomplicating the generation.

I'm also happy with the ability to regenerate identical terrain using Scriptable Objects. This forms the basis of a multiple island system extension, as Scriptable Objects are easily serializable. The terrain textures are however bounded within certain heights. This is, in part, unrealistic, and would be visually alarming had I not blended the textures together. Despite this, the terrain is diverse and looks reasonably realistic, however I would like to consider some alternate approaches or improvements to resolve this issue.

The introduction of natural objects onto the terrain certainly makes the world feel more engaging and detailed. The PDS algorithm is great in distributing the trees in realistic areas and emulating their natural separation. However, despite the random nature of the algorithm, from the player's perspective, the trees appear rather uniform, and the world can appear quite bland. This was, to an extent, governed by my choice of assets, as I felt that no free assets on the Unity store would fit the style of the game. Nevertheless, I would be interested to implement an algorithm to decrease the uniformity of the natural game objects spawned using the PDS algorithm.

I think that the camera system was perhaps the most effective part of my solution, as the movement of the camera feels extremely smooth and its manoeuvrability doesn't look out of place. The camera suitably allows the player to notice enemies, such that they can avoid them if necessary. I had hoped to add focus bars when the player repositions the camera behind themselves, however this feature is only offered by Unity Pro, and I didn't want to focus my time on my own version, as my aim was to implement the objectives to the best possible quality. The enemies themselves have an effective behaviour, as their patrol is random, and their approach to and attacks on the player are balanced. I think that the enemies would be more interesting if they had a more detailed model and animations, however I didn't consider this to be a priority of the project, as it doesn't contribute to game fundamentals. By simply using cubes, I was able to focus more on the behaviour of enemies, rather than their appearance.

All the collectables items, I believe, provide an interesting element to the game while it remains in a sole island form and meet the desired objective. They exhibit the idea that any exploration of the island can be beneficial, as although the player may have no idea where the goal or they might be, exploring will most likely provide them with information to help them. I'm very happy with the appearance of the UI message when these items are collected, as they suit the style of the game. The pathfinding works effectively to avoid movement up hills as well as avoid enemies where possible. However, there are factors that can cause the pathfinding to become quite expensive and stop the game for a short while. This occurs on large maps, where more nodes need to be navigated; when the goal object is place by an enemy, and the algorithm will be reluctant to approach through the enemy's FOV; and when goal object can only be reached by steep ascent. These are issues that I am confident I could resolve in a greater time-span and I have considered how I could approach them. Despite this, the pathfinding algorithm works very well in regard to its purpose, such I feel that my cost functions and coefficients have been well defined and adjusted.

END USER FEEDBACK

I have copied in the final feedback email I received from B.Evans having showed him the completed solution:

Tom,

After looking at your program I have a few thoughts and suggestions:

The enemies seem a bit weak. The only way I can see you dying to one is by standing still and waiting for it to chip down your health. I think perhaps they need to do more damage and also probably a fire rate increase. However, they look great and they can easily spot you and effectively navigate to you.

On larger maps, it can take a very long time to find the objective or any pickups in the game. Either the map needs to be smaller or more pickups need to be available. It would be nice if some objects spawn closer to the player so they are easier to find, whilst the goal remains far away. Alternatively, there could be a combat-system so the game is still engaging in the meantime while looking for the pickups, as finding enemies is less of an issue.

Once the pathfinder had been collected, you still have to enter the pause menu every time you want to check the direction. It would be nice to have it appear in the minimap in the corner of the screen so you can follow it while moving, because otherwise the game doesn't flow as well.

The program runs very smoothly, the terrain generated looks realistic and natural and I like the map falloff which makes it seem like an island. Perhaps finding alternative methods of determining the terrain texture would allow for increased variation in terrain types, however despite this, after viewing many iterations of maps, each one appears unique and impressive. Such, I feel that it would be nice to see some form of visualisation of the map generation while the game loads.

The camera movement looks great, is smooth and avoids obstacles very well, but it would be good to have an option to invert the camera, as by default inversion is enabled in the vertical direction and I found it difficult to control.

Overall, the game looks brilliant and the low poly style works very well. You've clearly adjusted the settings in the noise generation to produce a natural looking map and it shows.

Ben

SUMMARY OF END-USER FEEDBACK

The initial comment was about the behaviour of enemies. I agree with all of the points that my end-user has made in this area and will try to improve them where possible. He commented on the attack of the enemies, stating that the projectiles were fired too low, too infrequently, and that they dealt too little damage. The result of changing this is that approaching enemies would be more dangerous, and so obtaining the 'Enemy Finder' would become more significant. He later noted how the enemies would instantly turn away when the player became undetected. He thought that it would be interesting for the enemies to have an additional behaviour that they follow in the moments after losing sight of the player. I certainly think that this is a good suggestion, but I was unable to implement the change in the time-span.

He shared similar thoughts to my own with regard to the world generation. He was very impressed by the appearance of the world and liked the use of the falloff-map to create an island. He thought that it would be interesting to see a visualisation of this or one of the other algorithms in the game as a loading screen whilst the map loads. Like me, he thought that the game would benefit from a greater range of natural game objects and terrain types. He said that he is interested to see how I would employ an improved variety of terrain types without solely using height, whilst keeping the appearance of the map realistic. He felt that larger maps could feel quite empty as the number of collectable objects doesn't change but mentioned that this wouldn't be a problem were there a greater range of items or combat capabilities.

My end-user thought that the game had great potential and noted that a combat system would be the most desirable feature for him to see. Despite this, he found the collectable items made the game more interesting and purposeful but suggested some improvements that could be made to their placement. He thought that I could place the goal collectable further than a certain radius of the player, and the collectables within a shorter range. I'm really happy with this suggestion, as it allows items like the compass to be spawned close to the player, which suits the behaviour of the compass, as it isn't extremely influential, but it is helpful to find it earlier in the game. With regard to the pathfinding object, he was very impressed by the route it showed, but suggested that it would be nice if there were some way of showing which direction the player should follow through an in-game feature.

In general, my end-user thought the movement and camera system were smooth and easy to use. However, he personally doesn't like using inverted cameras in games, and suggested that I add an option that would allow the player to change inversion in both x and y directions.

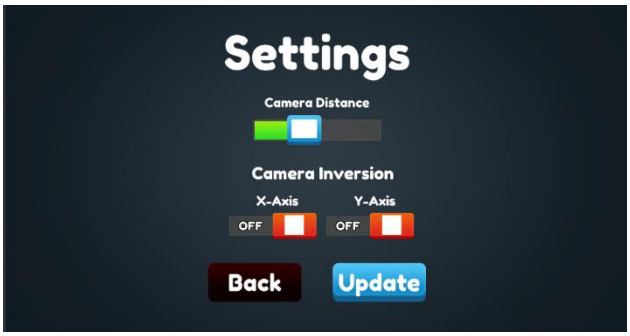
COMPLETED IMPROVEMENTS

With some of the issues that the end-user and I found with the project, there were solutions that I felt I could implement in my given time-span. I began by addressing B. Evans' concerns over the enemies, and such I have doubled their fire rate, and increased the target to around the player's chest height. I have also increased the damage of each projectile from 5 to 30.

To improve the nature of the collectable objects in game, I have added a spawn radius or annulus to each one's dynamic creation. I have done this by re-using the annulus function I created to aid in my Poisson Disc Sampling algorithm. This means that I can define the distance that the objects should spawn from the player, preventing the goal collectable or pathfinder from being found too nearby. For the pathfinder object, I have removed the influence of any enemy whose FOV spans the goal collectable, this is because the player would have to pass through it anyway, and so it shouldn't determine the path taken, and as a result dramatically slow down the algorithm. This change has made the speed of the pathfinding much more consistent, but I still feel that there are slight improvements that could be made.

I have also added an option in the menu that allows players to select whether they would like to use camera inversion in either or both axes. This is the type of improvement that I would need to implement in order to make the game more serializable and reach some of the extension objectives.

Settings Menu:



This image shows the new settings menu that I have developed as a result of the end-user's feedback. The menu allows the user to change the standard camera distance, as well as decide whether they want camera inversion in either axis. To do this I have used Unity's PlayerPrefs feature that allow me to both save and load values between game sessions. This is ideal for serialising small amounts of data, but if I were to develop more advanced settings options, then I would likely create a file to store the data.

POTENTIAL IMPROVEMENTS

Having evaluated the project myself and received my end-user's feedback, I have considered some of the success and areas for improvement within the project. Beginning with the terrain generation, I think that it's possible to make almost endless additions and variations in this field. For example, I think that it would be great to analyse the terrain in more ways that just the height, such as using the angularity or roughness of the terrain to determine a biome. I could implement a Voronoi Graph to split the terrain into multiple random sections, before deciding the most suitable terrain type for each section. This would be an interesting way of improving the randomness of terrain between separate generations.

In terms of object placement, I'm sure that many improvements could be made. Firstly, I would hope to introduce a wider range of natural GameObjects, as although I met my objective, I feel that a greater range of objects would improve the aesthetic of the game and provide some interesting challenges in terms of how to place them realistically. To do this, it may possible to use a Voronoi Graph again, to distinguish areas and provide each with different spawning coefficients. This would hopefully resolve the issue of the bland uniformity of natural objects in the current solution.

For the enemies in game, I would try to improve their variety in the future. To do this I would add some randomness to the values used in their dynamic creation. Different tiers of enemies would be of varying strength and have different FOV's. This would provide the game with a more linear structure, as player's might not be able to encounter powerful enemies until; they had explored other areas to become stronger. In some of the existing system's that I researched, the enemies had a detection gauge. This allowed for variety in the rate that detection occurred, based on distance, sound and the stealth values of the player. I think that this would look quite impressive and provide some interesting interactions with enemies. In addition, as mentioned previously, I would try to improve the appearance of the enemies, although I would need to acquire some considerable skill in animation to warrant doing this.

I added an extension to the camera such that the player can focus in behind themselves at any time, this was reminiscent of some of the existing systems features. Should I upgrade at any point to Unity Pro, I would definitely add focus bars to the game if it were suitable with the combat system. Alternatively, without time constraints, I would be happy to spend time implementing this myself. Furthermore, the ability to switch the focus of the camera between the player and enemies would be a great addition should a combat system be added.

The addition of an effective combat system would drastically improve the quality of the game. However, I felt that with my current knowledge of animation, and limited timespan, I would only waste time implementing an ineffective system. The potential that a combat system offers would make the program extremely interesting, as different terrain areas could house enemies that fight in different ways, or weapons that have different capabilities. This would give the player incentive to navigate between multiple islands throughout the game, giving the game a greater sense of progression and purpose.

UI in a similar way has almost infinite range for improvements, and its development would also stray me from the fundamentals defined in my objective. Despite this, I realise that an impressive UI is important in a successful game, and I would love to see interaction with in game items in the inventory menu.

I think that introducing the extension objective to revisit islands would be paramount if I were continue development of the project. This would make the game more 'replay-able' and would introduce the possibilities for different islands to tend towards different terrain types, as seen in some of the existing systems that I discussed in analysis. Thus, unique items and enemies could be found on certain islands, giving the player the incentive to explore as much as possible.

I would like to add a greater range of collectable objects like those mentioned in my extension objectives. This would add greater variety to the game and hopefully make it more interesting to play. With a greater variety of objects, it would make it much easier to implement island unique items as mentioned above, and to introduce a varying combat system depending on the player's weapon.

Despite this, I am very pleased with all that the project has achieved, in meeting my defined objectives, extending in ways that I didn't fully expect in design, and piquing my interest in the challenges it provided. Thus, I deem the project to be fairly successful and am intrigued by some of the possible improvements that I can make.

Bibliography

TERRAIN GENERATION

RedBlobGames. *Making maps with noise functions.* 2015. Available from: https://www.redblobgames.com/maps/terrain-from-noise/

YouTube: Sebastian Lague. *Procedural Landmass Generation.* 2016. Available from: https://www.youtube.com/playlist?list=PLFt_AvWsXloeBW2EiBtl_sxmDtSgZBxB3

YouTube: TheHappieCat. *How Procedurally Generated Terrain Works.* 2016. Available from: https://www.youtube.com/watch?v=JdYkcrW8FBg

YouTube: Brackeys. Procedural Terrain in Unity – Mesh Generation. 2018. Available from: https://www.youtube.com/watch?v=64NblGkAabk

OBJECT PLACEMENT

Bridson R. *Fast Poisson Disk Sampling in Arbitrary Dimensions.* 2007. Available from: https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf

YouTube: The Coding Train. *Poisson-Disc Sampling.* 2016. Available from: https://www.youtube.com/watch?v=flQgnCUxHlw

PATHFINDING

RedBlobGames: Amit Patel. *Introduction to the A* algorithm. 2014, 2016.* Available from: https://www.redblobgames.com/pathfinding/a-star/introduction.html

RedBlobGames: Amit Patel. *Amit's Thoughts on Pathfinding.* Available from: http://theory.stanford.edu/~amitp/GameProgramming/index.html

RedBlobGames: Amit Patel. *Movement Costs for Pathfinders.* Available from: http://theory.stanford.edu/~amitp/GameProgramming/MovementCosts.html

Nash, Daniel, Koenig, Felner. *Theta*: Any-Angle Path Planning on Grids.* Available from: https://web2.qatar.cmu.edu/~gdicaro/15381/additional/theta-star-AAAI-07.pdf