Glasgow Caledonian University

Department of Applied Computer Games (DACG)

Honours Project - MHW225671

FINAL REPORT

2021-2022

Submitted for the Degree of:

BSc Computer Games (Software Development)

Project Title:	Procedurally Generated World Map With Distinct Biomes and Cultural Regions
Author:	Mark Flanagan
Project Supervisor:	Hamid Homatash
Word Count:	10,256

(**Word count *excludes contents pages, figures, tables, references and Appendices***)

"Except where explicitly stated, all work in this report, is my own original work and has not been submitted elsewhere in fulfilment of the requirement of this or any other award"

Signed by Student: _____ **Date: 12/04/2022**

GitHub Repository: https://github.com/SwanDiveGames/honours-project

Final build:
https://simmer.io/@SwanDiveGames/honours-random-world-generation-w-cities-named-by-region

Abstract

Procedural Content Generation (PCG) is the method of generating game content, such as maps, levels, names, quests or characters algorithmically. This process adds to the replayability of a game experience whilst simultaneously lessening development costs, making it well-suited to smaller development studios seeking to create larger experiences.

PCG methods often employ pseudo-randomisation techniques, in which content is randomised with constraints in order to ensure that any generated content maintains the core design intent. Oftentimes, these constraints can lead to generated content seeming too familiar on repeated runs, or too random. For the purposes of world maps for games, most notably used for tabletop roleplaying games (TTRPGs) or computer roleplaying games (cRPGs), these either lead to maps being mono-cultural in their naming conventions (that is, all areas of the map have names originating from the same cultural influence, usually from Western fantasy), or being entirely randomised (names from Western and Middle-Eastern fantasy appearing directly beside each other in a way that doesn't seem natural, for example).

This project introduces a method by which multiple different cultural names can be generated based on the region of the map they appear in, with the purpose of grouping these points of interest into more distinct cultural regions to give a sense of natural emergence in the generated world. A world map is generated by tiling multiple terrain objects in the Unity game engine and overlaying a series of Perlin and uniform noise waves to generate height, heat and moisture maps and combine them into distinct biomes (e.g. savanna, grasslands, tundra etc.). When points of interest are instantiated into the map, they check the coordinates they are in for biome data and name themselves accordingly using a pseudo-random name generator which has distinct naming conventions for each biome.

The map generation method proved moderately successful in these tests, with 73.9% of 23 respondents agreeing that the generator was effective. Respondents did note patterns in groupings of names, suggesting a successful distinction between cultures, but it should be noted that this was not consistent across all generated maps, and respondents noticed repetition in names on a somewhat regular basis. With future refinement, it is believed that this tool could produce consistent, viable world maps with diverse points of interest.

Acknowledgements

Thanks to Hamid Homatash, the advisor of this project, who offered excellent guidance and made this possible, for his continued support throughout this year and last.

Thanks to Alan Jack, my lecturer, who gave great design insights and useful resources.

Thanks to my family, who have supported me through many years of study to reach this point.

Thanks to Sarah McNeil for encouraging me and keeping me focused close to the end.

Thanks to Aaron Jeffers, who has helped immeasurably throughout my degree.

Thanks to Alessio Scisci, the best student representative we could have asked for.

And thanks to my friends, whose constant reminders I am near the finish line helped me make it all the way.

Contents

1. Introduction

1.1 Background

Procedural Content Generation (PCG) is the algorithmic generation of game content with limited or no human contribution (Togelius et al, 2013). More specifically, for the purposes of this project at least, game content refers to the elements within a game that the player interacts with (maps, items, quests, music, characters etc.), but not the game itself (game engine, artificial intelligence, game mechanics). A PCG *system* refers to a system that incorporates a PCG method as one of its parts (Shaker et al, 2016).

Although algorithmic in nature, PCG is actually rooted in design, not development, as it essentially aims to supersede or otherwise lighten the workload of games designers by supplementing their work, or replacing it entirely, based on the needs of the product. This, of course, isn't to say designers will no longer be needed - their input is essential to the implementation of any PCG algorithm. Whether intentional or not, every PCG system implicitly encodes a formal theory for both the game design process and the product that is being procedurally created (Smith, 2015). Procedurally generated content still needs to be designed and scaled, and is an art in itself (Berente, 2021). Indeed, Derek Yu, creator of indie classic *Spelunky*, said "it was my distinct lack of passion for programming which led me to the idea" in reference to his own map generation algorithm (Yu, 2016). The desire to automate the design process is both one of economic benefit and design benefit, as less hand-made content is required, and there is great potential for a longer-lasting play experience if PCG algorithms are made well.

1.1.1 Procedural Map Generation

A lot of commercial games, like *Spelunky, Darkest Dungeon II,* and other so-called "*Rogue*-likes", employ pseudo-randomised map generation with new layouts, enemy and loot spawns, ally spawns, and randomised objectives for each "run" of the game. This level of pseudo-randomisation - the random selection of assets from a predefined list to create a greater number of unique combinations - is also present in exploration-based titles such as *Minecraft* and *Terraria* in which entire worlds are generated and can be saved to play in. Procedural map generation can also be used for static, more tailored worlds however - famously, *Bethesda* used procedural generation to create the landscape of *The Elder Scrolls IV: Morrowind* (which admittedly led to some oddities, such as the "*Golden Coast"* region actually being a grassy landscape), and also plan to create the map for the new untitled sixth game procedurally (GameRant, 2020).

There are two different types of map generation: "space" or "system" generation. "Space" generation is where a world as a relative blank slate is created, with the intent for users to enact their own designs upon the world. "System" generation is where a world is generated as a complex backdrop for the player to interact with the game's systems, such as factions, economy etc. (T Short, T Adams, 2017). These two methods have unique design requirements, and therefore their algorithms can appear quite different. "Space" generation requires more variation in topography and available resources, altering which of these a player can interact with, exploit and ultimately use in their own designs (such as in games like *Civilization* or *They are Billions,* in which a blank canvas "space" world is generated and the player expected to build their empire within. Strategies for success can vary wildly based upon the available resources in the player's area). "System" generation, meanwhile, has less of a focus on varied topography (but still some expectation of variation) and more on the generation of subsystems and points of interest for the player to interact with (cities, towns, merchants, dungeons etc.). In TTRPGs such as *Dungeons & Dragons*, or Rogue-like games such as *Stoneshard*, the

topography can pose a challenge, but that challenge is secondary to the player's primary interactions with the world - fighting monsters, completing quests for non-player characters (NPCs), increasing reputation with factions and figures in the world, and levelling up their character through these interactions.

1.1.2 Problem with True Randomisation for Maps

As described in the previous *Elder Scrolls IV: Oblivion* example, and present in other randomly generated mapping tools (particularly in dedicated generation tools for TTRPGs such as *Hexographer*), oftentimes a regional point of interest can be generated which does not match the expected naming conventions of the surroundings. Games and mapping tools have some solutions to this problem, but these solutions can still impact the verisimilitude of the world the player is engaging in. *Stoneshard* and *Terra Randoma*, for example, limit the scope of their generated worlds to a single cultural region or influence. This means that all generated names come from the same cultural origins, and therefore do not seem out of place when juxtaposed with other points of interest in the area. However, this limits the scope of the maps to smaller areas, and reduces (or negates entirely) the player's ability to engage with factions of other cultures (for example, *Stoneshard*'s fantasy influence has it contain both Dwarven and Elven cultures, but neither of these feature in any way in the generated points of interest the player can explore). This can also lead to these names seeming more generic, as they all stem from the same background. Other tools, such as *Azgaar's Fantasy Map Generator*, do produce multi-cultural worlds with distinct regions, but oftentimes these cultures can appear in areas one would not expect them to appear in our own world (for example, East-Asian fantasy cultures appearing in deserts, or in snowy regions, and directly beside Western fantasy-influenced regions).

1.2 Research Question

How can procedural content generation techniques be used to produce a world map with believable naming conventions for points of interest in distinct biomes?

1.2.1 Hypothesis

The development of a Perlin noise-based map system, utilising height, heat and moisture data to produce distinct biomes in which different cultures can exist, and using those biomes to generate cultural naming conventions, can produce a more believable, diverse, procedurally generated world.

1.2.2 Project Type

This project is a development-type project which will produce the map generator in question and evaluate its effectiveness in producing more realistic worlds through user testing and both qualitative and quantitative analysis.

1.2.3 Project Aim

The aim of this project is to utilise multiple facets of content generation to create a more complex, interesting piece of content, as opposed to simply focusing on individual pieces of generated content.

1.2.4 Justification

In procedurally generating any piece of game content, oftentimes developers focus on individual aspects of the content and generate them, then simply put pieces together. Although this method can produce a wide variety of content, it can often lead to more generic and less considered content being generated. By combining multiple content generation methods at the generation level, it is

believed that each individual piece of content will benefit from the other, and the final product will therefore be more interesting, believable, and unique in its output.

2. Literature Review

2.1 Previous Map Generation Work

Togelius and colleagues (2011) describe the difference between 'constructive' and 'generate and test' PCG algorithms, in which a constructive algorithm generates the content once, and is done with it, but ensures the content is fit for use during the construction phase, which can be done by only performing operations that are guaranteed to never produce broken content; whereas generate-and-test algorithms use both a generate and test mechanism. After content is created, it is tested according to set criteria, and if the test fails, all or some of the candidate content is discarded and regenerated until the content is good enough (Togelius et al, 2011).

In describing Search-Based Procedural Content Generation (SBPCG), they say it is a special case of the generate and test approach to PCG, but instead of simply accepting or rejecting candidate content, it grades it using a fitness, evaluation, and utility function. New candidate content is generated based on the fitness score of the previous candidate, and in this way the aim is to produce new content with a higher fitness value (Togelius et al, 2011). This method, usually specifically the 'evolutionary' SBPCG algorithm (Togelius et al, 2011), leads to content better fit for use while being more efficient than simply discarding 'failed' content as per standard generate-and-test algorithms.

In their paper on SBPCG, Togelius and colleagues (2010) defined the difference between necessary and optional content in a procedurally generated environment. They state; "necessary content is required by the players to progress in the game… whereas optional content is that which the player can choose to avoid… the difference here is that necessary content always needs to be correct… on the other hand, one can allow an algorithm that sometimes generates unusable weapons and unreasonable floor layouts if the player can choose to drop the weapon and pick another one or exit a strange building and go somewhere else instead" (Togelius et al, 2010).

This mirrors the design philosophy of *Spelunky*, as the creator described in his book of the same title (Yu, 2016), in which he describes the second and third rules of his level generation; "2. There must be a way to get from the entrance to the exit without using bombs, rope, or special items (anything not on this path can be walled off or otherwise inaccessible)... "3. The level generator must do its best not to create any places where the player can get stuck and have to use items to get out" (Yu, 2016).

Within each of these cases, it becomes clear that a generate-and-test algorithm is better suited to levels with a high amount of interactivity - that is levels with items, doors, enemies etc. that need to be built into the design and therefore can affect the viability of a given play space. In the case of a world map, however, there is little optional content, and the map primarily consists of necessary content, so these barriers do not exist. Instead, one is only interested in the generation of terrain and positioning of points of interest, which can be tested and confirmed in the generation stage without the necessity to test and iterate through fitness algorithms, thus saving computational power. To this end, the author intends to utilise a constructive algorithm, not a generate-and-test one.

Chen and colleagues (2020) created a similar algorithm to the proposed - a map generation technique in the Unity game engine which used model-based techniques to define the landscape, which itself is generated using Perlin noise techniques. Their worlds could have game-specific

constraints applied, and the produced game world could be utilised by developers and manually post-processed within Unity. In this case, Perlin noise created a viable landscape, and the Perlin noise used for the heightmap helped define the colour of terrain at certain areas. These techniques can be employed similarly for this project - constraints which can be altered by the developer within the game engine, and texture colouring based on the height of the region. However, this project seeks to go further with this concept, and can apply the colouring techniques to the different biomes created by the algorithm, as opposed to simply differentiating height.

In their survey of procedural content generation for games, Hendrikx and colleagues (2013) described Perlin noise and other pseudo-random number generation techniques as being easily mapped onto complex objects. This, they said, makes it common to use in the industry, and can imitate the level of detail that a larger resolution texture can provide. Perlin noise is significantly smoother than true random. The difference between numbers is gradual, and it is referred to as gradient noise (Kennard, S, 2019). This makes the produced noise much more pleasing to look at and well suited to maps.

Patel (2010) offered a different and unique approach to terrain generation in their Voronoi graph-based generation method. This generates a Voronoi graph of polygons, applying Lloyd's relaxation to smooth out the edges, and then use this data to generate a heightmap and apply textures accordingly. This method is very effective at creating islands with increasing elevation towards the centre, and includes differentiation between oceans and lakes as well as having some smaller islands off the coast of the main island. However, this method is primarily dedicated to the creation of a singular landmass with relatively restrictive rules, and the author themselves notes that other noise generation techniques may be more suitable for other games than their own focus. It does, however, employ a variety of biomes according to the Whittaker diagram (Whittaker, 1962), which proposes a model for natural communities divided into cross-sections between moisture and heat. This creates a natural appearance to the world's colouration and biomes, and thus this method can be employed.

2.2 Notes of Future Work

Togelius and colleagues (2013) described a theoretical game world generator which produced terrain, vegetation, roads, cities, people, creatures, quests, lore, dialogue, items and vehicles (Togelius et al, 2013). This would be an all-encompassing solution to procedural content generation, and would provide everything required for a game except for the underlying game engine itself. They additionally describe current PCG approaches as simply generating a single type of content, separate from others, and thus the products created appear generic and uninspired.

Van Der Linden and colleagues (2013) described online and 3D generation as open challenges, noting that generated dungeons are far from hand crafted ones (Van Der Linden et al, 2013). This reflects Togelius and colleagues'(2013) observation of the generic nature of generated content when produced outside the context of other content. Therefore, it is apparent that finer details are required to be produced in the generation stage to produce a realistic, believable and interesting piece of content,as opposed to after. Therefore, this project seeks to combine the generation of maps and points of interest so that both benefit from each other as opposed to contradict or ignore.

3. Execution

3.1 Terrain Randomisation and Noise Generation

3.1.1 Overview

The purpose of the project was to create a world map generator within the Unity game engine. The terrain generator should (1) enable developers to alter variables to change the map to their purposes; (2) it should generate a feasible terrain; (3) it should produce a unique map on every iteration; (4) it should produce distinct biomes correlating to Whittaker's diagram; (5) it should store data for biomes at certain coordinates.

To these ends, the generator employed multiple randomised (3) Perlin noise waves to produce a heightmap, heatmap and moisture map. These were overlaid on a series of terrain tiles, whose number could be altered by developers to create larger or smaller maps (1). The terrain's shape would be altered according to the heightmap to produce hills, mountains, valleys etc. (2), and the texture applied would use a combination of the height, heat and moisture maps to produce biomes as a mixture of heat and moisture and colour the terrain accordingly (4). Data for biomes would be stored as coordinates, allowing for algorithmic distinction between biomes, for use in name generation later (5).

3.1.2 Height Maps

Terrains in the world have a wide variety of heights associated - mountains, hills, plains, valleys and the ocean all have varying heights. In order to replicate this effect for the world map to be generated, this project would employ Perlin noise. Perlin noise is easily mapped onto complex models, and Unity contains built-in functions for Perlin noise. Although a single Perlin noise wave can produce areas with jagged appearance, this project would combine a multitude of waves in order to create a smoother, more realistic landscape for the terrain objects.

Firstly, a script was created for the generation of noisemaps, simply called "NoiseMapGeneration". Within this script, the "GeneratePerlinNoiseMap" function was created (see Fig.1). This function had parameters for tile width, tile depth (labelled as map width and depth respectively), and scale, as well as the X and Z offset of each tile, and an array of randomised "waves". Waves were for the creation of Perlin noise waves and had float variables for seed, frequency and amplitude. Note that the waves were randomised before the noisemap is generated to ensure that the Perlin noise map is consistent across all terrain tiles, thus creating a cohesive landscape across all tiles. Randomising waves per tile would produce a different pattern on each tile with no cohesive connections between, thus failing to qualify for criteria (2).

For the creation of a perlin noise map, an array of float coordinates was initialised and a for loop created. This for loop was designed to iterate through every X and Z coordinate in each tile, and so each was bound by the tile's width and depth respectively. Initially, variables sampleX and sampleZ were created by taking the input X and Z coordinates, applying the tile's offset on X or Z and then dividing by the desired scale of the noise map (a lower scale means larger landmasses).

A float, noise, was initialised as 0.0f, and then a Perlin noise was generated using the sampleX and sampleZ, each multiplied by the wave frequency and then added to the seed as an offset on the Perlin wave. A normalisation value was then created and the wave's amplitude was added. This process was repeated for each height wave that the algorithm generated by randomly assigning a

value for the seed (see Fig.2), thus ensuring that each wave is unique and the combination of waves moreso, making each map's heightmap entirely unique (meeting criteria 3). Seed values of 2500 and 7500 were chosen through iterative testing as producing maps of a consistent quality, however these values could be altered to any between 0 and 9999.

```
//Function to generate a noise map using Perlin noise. Creates a float at each coordinate on the terrain
//as a number between 0 and 1 to give that coordinate a random height.

3 references
public float[,] GeneratePerlinNoiseMap(int mapDepth, int mapWidth, float scale, float offsetX, float offSetZ, Wave[] waves)
{
    //Create an empty noise map with the mapDepth and mapWidth coordinates (Z and X respectively)
    float[,] noiseMap = new float[mapDepth, mapWidth];

    //Simple for loop to iterate each (X, Z) co-ordinate to apply a noise value.
    //Assigns a Z value, then iterates through all X values, then iterates Z and repeats.
    for (int zIndex = 0; zIndex < mapDepth; zIndex++)
    {
        for (int xIndex = 0; xIndex < mapWidth; xIndex++)
        {
            //Calculate sample indices based on the coordinates and scale
            float sampleX = (xIndex + offsetX) / scale;
            float sampleZ = (zIndex + offSetZ) / scale;

            //Code for single Perlin noise wave

            ////Generate noise value using perlin noise
            //float noise = Mathf.PerlinNoise(sampleX, sampleZ);
            //noiseMap[zIndex, xIndex] = noise;

            //Using multiple waves
            float noise = 0f;
            float normalization = 0f;

            foreach (Wave wave in waves)
            {
                //Generate noise value using Perlin Noise for a given wave
                noise += wave.amplitude * Mathf.PerlinNoise(sampleX * wave.frequency + wave.seed, sampleZ * wave.frequency + wave.seed);
                normalization += wave.amplitude;
            }

            //Normalise the noise value so that it is within 0 and 1
            noise /= normalization;

            noiseMap[zIndex, xIndex] = noise;

        }
    }

    //Return the generated coordinates
    return noiseMap;
}
```

Fig.1 GeneratePerlinNoiseMap function within the NoiseMapGeneration script

```
//Randomise waves
for (int i = 0; i < heightWaves.Length; i++)
{
    heightWaves[i].seed = Random.Range(2500, 7500);
    heightWaves[i].amplitude = 1;
    heightWaves[i].frequency = 1;

    Debug.Log("Heightwave" + i + " randomised");
}

for (int i = 0; i < heatWaves.Length; i++)
{
    heatWaves[i].seed = Random.Range(2500, 7500);
    heatWaves[i].amplitude = 1;
    heatWaves[i].frequency = 1;

    Debug.Log("Heatwave" + i + " randomised");
}

for (int i = 0; i < moistureWaves.Length; i++)
{
    moistureWaves[i].seed = Random.Range(2500, 7500);
    moistureWaves[i].amplitude = 1;
    moistureWaves[i].frequency = 1;

    Debug.Log("Moisture wave" + i + " randomised");
}
```

Fig.2 Wave Randomisation

The noise variable was then divided by the normalisation to ensure it was a value between 0 and 1, and then the noiseMap value for the Z, X coordinate was saved as that value. Once the algorithm had iterated through all coordinates in the tile, it returned the noiseMap array.

In order for the heightmap to display on the map tiles, the terrain meshes themselves had to be altered and the Y coordinates adjusted to match the Perlin noise values. To do this, a function UpdateMeshVertices was created with the noisemap as an input variable (see Fig.3).

```
1 reference
private void UpdateMeshVertices(float[,] heightMap)
{
    int tileDepth = heightMap.GetLength(0);
    int tileWidth = heightMap.GetLength(1);

    Vector3[] meshVertices = this.meshFilter.mesh.vertices;

    //iterate through all the heightMap coordinates, updating the vertex index
    int vertexIndex = 0;
    for (int zIndex = 0; zIndex < tileDepth; zIndex++)
    {
        for (int xIndex = 0; xIndex < tileWidth; xIndex++)
        {
            float height = heightMap[zIndex, xIndex];

            Vector3 vertex = meshVertices[vertexIndex];

            //Change the vertex Y coordinate, proportional to the height value
            meshVertices[vertexIndex] = new Vector3(vertex.x, this.heightCurve.Evaluate(height) * this.heightMultiplier, vertex.z);

            vertexIndex++;
        }
    }

    //Update the vertices in the mesh and update its properties
    this.meshFilter.mesh.vertices = meshVertices;
    this.meshFilter.mesh.RecalculateBounds();
    this.meshFilter.mesh.RecalculateNormals();

    //Update the mesh collider
    this.meshCollider.sharedMesh = this.meshFilter.mesh;
}
```

Fig.3 UpdateMeshVertices function

Each coordinate of the heightmap was input as a [Z, X] coordinate, and so the depth of each tile was the sum of the Z coordinates and the width is the sum of the Xs, and so tileDepth and tileWidth become heightmap.GetLength(0) and (1) respectively. An array of vertices was created (with vertices being represented as 3 dimensional coordinates, Vector3s in this case), and the vertices of the terrain tile's mesh were stored. Then, quite simply, for each [Z, X] coordinate on the tile, the Y coordinate associated with it was evaluated as the value of the Perlin noise map at that coordinate, then multiplied by a variable height multiplier. This height multiplier was able to be set by developers to satisfy criteria (1). This is applied via a for loop to iterate through all vertices in the mesh, and then the mesh filter and collider were both updated to match.

After this, data for terrain types could be stored in the coordinates. This was a precursor to the Biome textures, and simply differentiated the environment into three environment types based on height - water, grass and mountain. To represent this, a new class was created as part of the TileGeneration script; TerrainType. This class contained four variables: name, a string which contains the name of the terrain type (in the heightmap example, it was only water, grass or mountain, but for later biomes it was the biome's name); threshold, a float containing the value under which the terrain type exists (in the current project, all height values at 0.4 or below are water, 0.4-0.7 are grass, and 0.7 to 1 being mountain, and so the water threshold was 0.4, grass 0.7 and mountain 1); colour, a color the developer can manually assign to each terrain type (for these terrains the colouring was more obvious, with water being a shade of blue, grass a green and mountain a brown. For later biomes,

one can replicate the real world colour or be more creative and set any RGB colour to satisfy criteria (1)); index, an int storing a table index for the terrain type (this became relevant for biome data, it was not relevant for the heightmap specifically). An array of terrain types called heightTerrainTyes was created in the TileGeneration script, and the script applied as a component of the terrain tile prefab (see Fig.4), in which the colours and thresholds can be modified. Note also the inclusion of an animation curve, called Height Curve, which serves to smooth the transition between terrain types, creating an area of variance between the two colours (these were previously harsh boundaries and appeared unnatural).

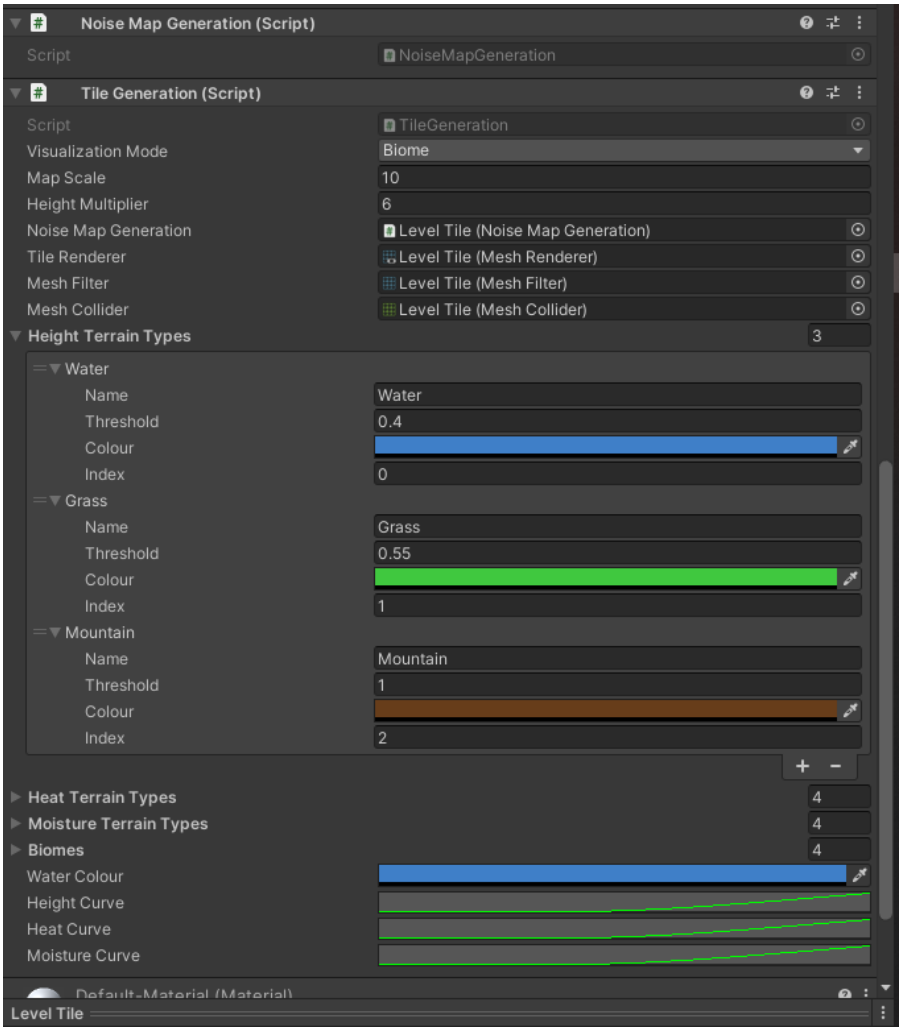


Fig 4. Tile Prefab with Heightmap colours and thresholds

The last step was to produce this as a texture on the tile. To do this, the BuildTexture function was used, which takes in a height, heat, or moisture map (based on which the developer wishes to display on the map. Note that in the code the input is labelled heightMap but this is simply a coordinate array which is any of the Perlin noise maps required), the terrain types of the desired map, and an empty array of terrain types for each coordinate in which terrain type data can be stored for the biomes. Then (in the case of heightmaps) it checked the Perlin height value against the thresholds and assigned the appropriate colour for every coordinate on the tile, then updated the tile's texture with the new colour array (see Fig 6.).

```
3 references
private Texture2D BuildTexture(float[,] heightMap, TerrainType[] terrainTypes, TerrainType[,] chosenterrainTypes)
{
    int tileDepth = heightMap.GetLength(0);
    int tileWidth = heightMap.GetLength(1);

    Color[] colourMap = new Color[tileDepth * tileWidth];
    for (int zIndex = 0; zIndex < tileDepth; zIndex++)
    {
        for (int xIndex = 0; xIndex < tileWidth; xIndex++)
        {
            //Transform the 2D map index into an array index
            int colourIndex = zIndex * tileWidth + xIndex;
            float height = heightMap[zIndex, xIndex];

            //Choose a Terrain Type based on the height value
            TerrainType terrainType = ChooseTerrainType(height, terrainTypes);

            //Assign the colour according to the terrain type
            colourMap[colourIndex] = terrainType.colour;

            //Save the chosen terrain type
            chosenterrainTypes[zIndex, xIndex] = terrainType;
        }
    }

    //Create new texture and set its pixel colours
    Texture2D tileTexture = new Texture2D(tileWidth, tileDepth);
    tileTexture.wrapMode = TextureWrapMode.Clamp;
    tileTexture.SetPixels(colourMap);
    tileTexture.Apply();

    return tileTexture;

}

1 reference
TerrainType ChooseTerrainType (float noise, TerrainType[] terrainTypes)
{
    //for each terrain type, check if the height is lower than the terrain's height
    foreach (TerrainType terrainType in terrainTypes)
    {
        //return the first terrain type whose height is higher than the generated one
        if (noise < terrainType.threshold)
        {
            return terrainType;
        }
    }

    return terrainTypes[terrainTypes.Length - 1];
}
```

Fig 5. BuildTexture and ChooseTerrainType functions

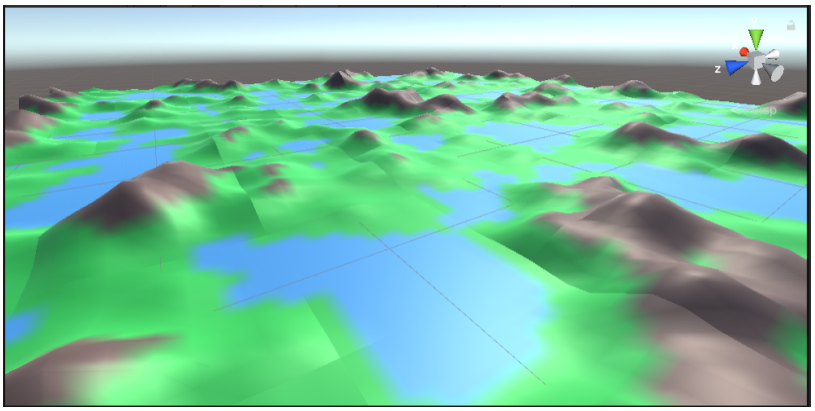


Fig 6. A generated map with 10x10 tiles and rendering height terrain types

3.1.3 Heat and Moisture Maps

For the purposes of this project, moisture maps were produced in much the same way as the heightmaps, being a series of Perlin noise waves multiplied and evaluated together to produce a smoother output (see Fig 2. and Fig 10.). However, the terrain types for moisture were different, in that there were four types: dryest, dry, wet and wettest (see Fig 7.). Again, these thresholds and colours were changeable to satisfy criteria (1). The method of generation and texture creation was identical to height maps.

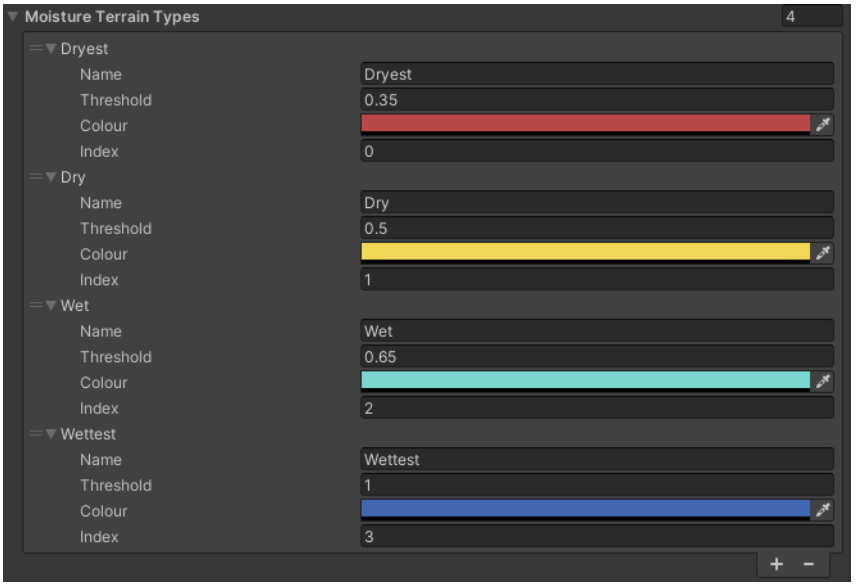


Fig 7. Moisture terrain types and their thresholds

However, heat maps are slightly different. On Earth, there is an equator and poles with wildly different temperatures, the temperature is warmest around the equator and coldest at the poles, with heat dispersed more uniformly across this pattern (with colder temperatures as one travels from the equator to the poles). Additionally, temperatures tend to be warmer lower down and colder higher up.

In order to represent the equator and poles, a Uniform noisemap was created (see Fig 8. and Fig 9.) by setting the noise value equal to the distance from the centre of the tile.

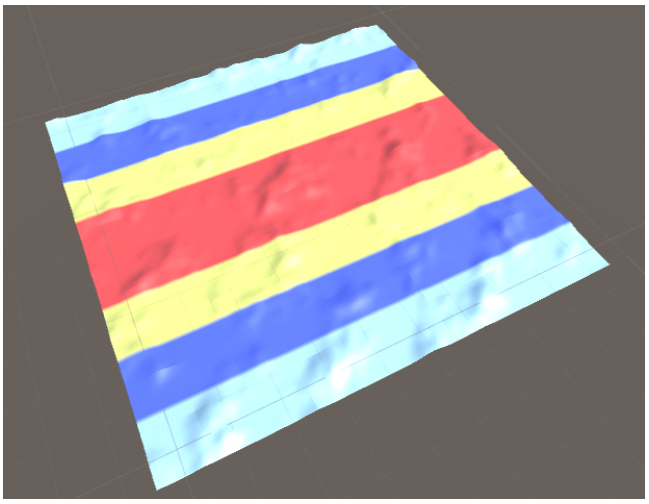


Fig 8. The Uniform Heat Map before noise is applied

```
//Function to generate a noise map for heat distribution for generating biomes. Using perlin nosie again to simulate uneven distribution (as is present in the real world)
//Z coordinate required this time to find the middle of the world and distribute heat accordingly (think hemispheres on Earth)
1 reference
public float[,] GenerateUniformNoiseMap (int mapDepth, int mapWidth, float centerVertexZ, float maxDistanceZ, float offsetZ)
{
    //Create an empty noise map with mapDepth and mapWidth coordinates
    float[,] noiseMap = new float[mapDepth, mapWidth];

    for (int zIndex = 0; zIndex < mapDepth; zIndex++)
    {
        //Calculate the sampleZ by summing the index and the offset
        float sampleZ = zIndex + offsetZ;

        //Calculate the noise proportional to the distance of the sample to the center of the level
        float noise = Mathf.Abs(sampleZ - centerVertexZ) / maxDistanceZ;

        //Apply the noise for all points with this Z coordinate
        for (int xIndex = 0; xIndex < mapWidth; xIndex++)
            noiseMap[mapDepth - zIndex - 1, xIndex] = noise;
    }

    return noiseMap;
}
```

Fig 9. GenerateUniformNoiseMap function

This created a simple, latitude-based heatmap, but didn't look very realistic. The real heat distribution of Earth is slightly more varied, and so the heatmap was then varied by multiplying values with multiple different Perlin noise waves (see Fig 2.). After this, higher regions were made colder, and lower warmer by adding the height values to the heatmap, thus ensuring that height varies the value and so alters its threshold (see Fig 10.). This ensured the heat distribution satisfies criteria (2).

```
//Generate a Heatmap using uniform noise
float[,] uniformHeatmap = this.noiseMapGeneration.GenerateUniformNoiseMap(tileDepth, tileWidth, centerVertexZ, maxDistanceZ, vertexOffsetZ);

//Generate a Heatmap using Perlin noise
float[,] randomHeatMap = this.noiseMapGeneration.GeneratePerlinNoiseMap(tileDepth, tileWidth, this.mapScale, offsetX, offsetZ, heatImport);
float[,] heatMap = new float[tileDepth, tileWidth];

for (int zIndex = 0; zIndex < tileDepth; zIndex++)
{
    for (int xIndex = 0; xIndex < tileWidth; xIndex++)
    {
        //Mix both heatmaps together by multiplying their values
        heatMap[zIndex, xIndex] = uniformHeatmap[zIndex, xIndex] * randomHeatMap[zIndex, xIndex];

        //Make higher regions colder by adding the height value to the heatmap
        heatMap[zIndex, xIndex] += this.heatCurve.Evaluate(heightMap[zIndex, xIndex] * heightMap[zIndex, xIndex]);
    }
}

//Generate a moisture map using Perlin noise
float[,] moistureMap = this.noiseMapGeneration.GeneratePerlinNoiseMap(tileDepth, tileWidth, this.mapScale, offsetX, offsetZ, moistureImport);
for (int zIndex = 0; zIndex < tileDepth; zIndex++)
{
    for (int xIndex = 0; xIndex < tileWidth; xIndex++)
    {
        //makes higher regions drier by reducing the height value from the heat map
        moistureMap[zIndex, xIndex] -= this.moistureCurve.Evaluate(heightMap[zIndex, xIndex] * heightMap[zIndex, xIndex]);
    }
}
```

Fig 10. Heat and Moisture Map Generation Code

The same BuildTexture function for height maps was used for both heat and moisture maps, and the tiles had a customisable field for selecting a visualisation mode to decide which texture is displayed on the map (thus further meeting criteria (1)) (see Fig 11.).

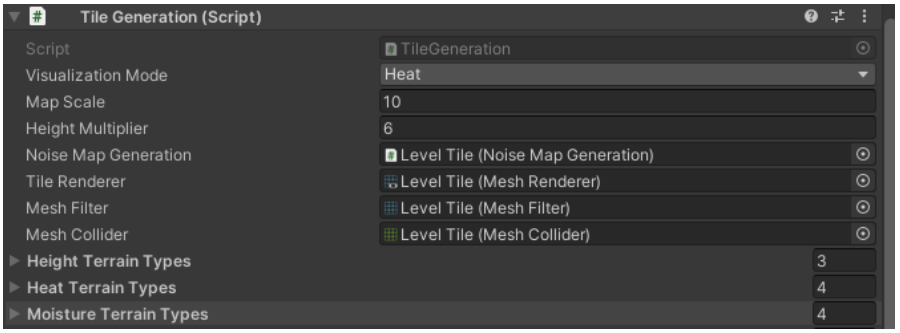


Fig 11. Visualisation mode

3.1.4 Biomes

For biome generation, a slightly simplified variant of Whittaker's model was deployed, making use of the four heat and four moisture terrain types. This gave a table of 16 entries (see Fig 12.) each with a different heat/moisture combined biome.

Moisture\Heat	Hottest	Hot	Cold	Coldest
Dryest	Desert	Grassland	Tundra	Tundra
Dry	Savanna	Savanna	Boreal Forest	Tundra
Wet	Tropical Rainforest	Boreal Forest	Boreal Forest	Tundra
Wettest	Tropical Rainforest	Tropical Rainforest	Tundra	Tundra

Fig 12. Biome Table

In order to represent this table within the Unity editor, two new classes were created: Biome, which contains name, a string, and colour, a color; and BiomeRow, which contains an array of Biomes representing a row in the table from Fig 12.. This was where the index attribute from terrain type became relevant, as the index was used later to access the biomes table and assign the correct biome.

To assign the biomes correctly, the BuildBiomeTexture function was created (see Fig 13.), which took the designated terrain types from the other built textures as input (see Fig 5.), as well as an empty coordinate matrix to store biomes. The algorithm iterated through all coordinates in the tile again, checking for terrain type data. Firstly, it checked if the coordinate was water. This became important later, as for the purposes of this project, water was not considered a biome and therefore no biome is assigned to these coordinates. Using the index of each terrain type, one could access first the moisture terrain type index for the correct biomes row, and then the heat terrain type for the correct column, thus selecting the correct biome for the moisture and heat combination in that coordinate.

```
1 reference
private Texture2D BuildBiomeTexture(TerrainType[,] heightTerrainTypes, TerrainType[,] heatTerrainTypes, TerrainType[,] moistureTerrainTypes, Biome[,] chosenBiomes)
{
    int tileDepth = heatTerrainTypes.GetLength(0);
    int tileWidth = heatTerrainTypes.GetLength(1);

    Color[] colourMap = new Color[tileDepth * tileWidth];

    for (int zIndex = 0; zIndex < tileDepth; zIndex++)
    {
        for (int xIndex = 0; xIndex < tileWidth; xIndex++)
        {
            int colourIndex = zIndex * tileWidth + xIndex;

            TerrainType heightTerrainType = heightTerrainTypes[zIndex, xIndex];

            //Check if the current coordinate is a water region
            if (heightTerrainType.name != "Water")
            {
                //If a coordinate is not water, its biome will be defined by the heat and mositure values
                TerrainType heatTerrainType = heatTerrainTypes[zIndex, xIndex];
                TerrainType moistureTerrainType = moistureTerrainTypes[zIndex, xIndex];

                //Terrain type index is used to access the biomes table
                Biome biome = this.biomes[moistureTerrainType.index].biomes[heatTerrainType.index];

                //Assign the colour according to the selected biome
                colourMap[colourIndex] = biome.colour;

                //Save biome in chosenBiomes matrix only when not water
                chosenBiomes[zIndex, xIndex] = biome;
            }

            else
            {
                //Water regions don't have biomes, they're simply water coloured
                colourMap[colourIndex] = this.waterColour;
            }
        }
    }

    //Create a new texture and set its pixel colours
    Texture2D tileTexture = new Texture2D(tileWidth, tileDepth);
    tileTexture.wrapMode = TextureWrapMode.Clamp;
    tileTexture.SetPixels(colourMap);
    tileTexture.Apply();

    return tileTexture;
}
```

Fig 13. BuildBiomeTexture function

Thus, one could select Biome for the visualisation mode, and the map generated distinct biomes based on the mixture of heat and moisture in the region, satisfying criteria (1), (2), (4), and (5) (see Fig 14.).

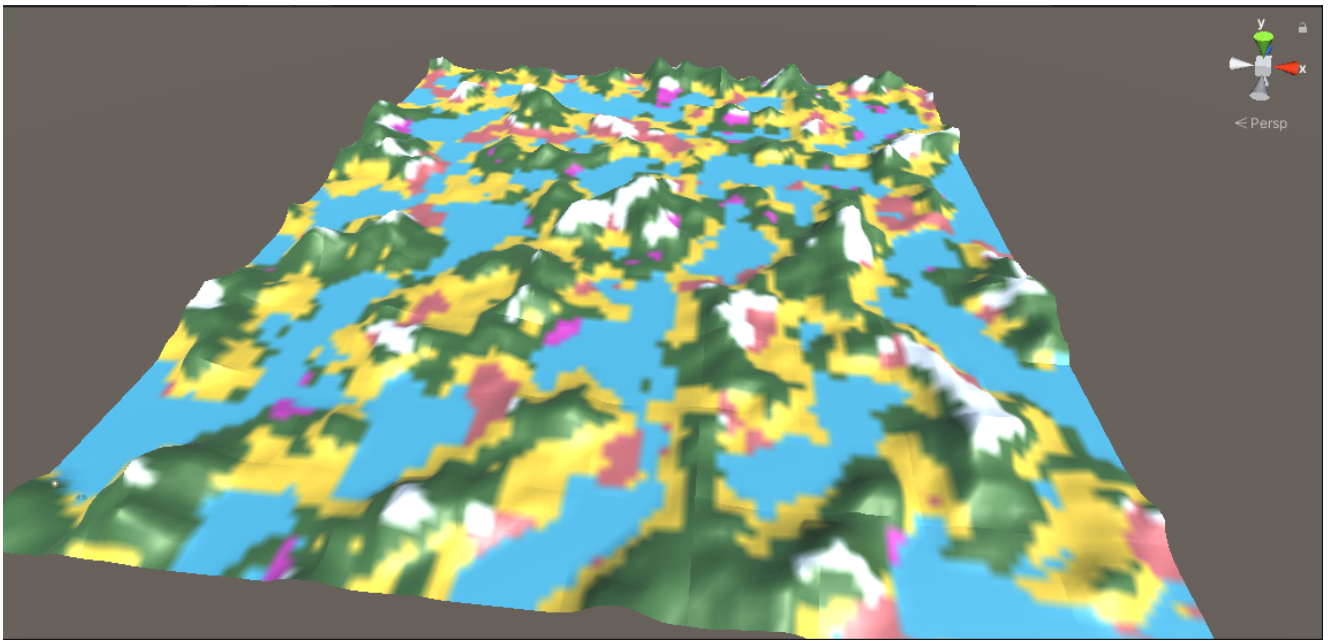


Fig 14. 10x10 generated world map with distinct biomes (note colours are set unnaturally for testing purposes and were later amended to reflect their natural state)

3.1.5 Rivers

The final component to terrain generation was to add rivers, which are an expected component of any world map, and thus complete the satisfaction of criteria (2). On Earth, rivers flow from high areas to low areas until they reach water, and this could be replicated algorithmically using stored heightmap data identifying regions of water.

In order for rivers to flow between tiles, and create a more cohesive map, the algorithm must first be able to see all tiles at the same time, and have access to tile data. To this end, the TileData class was created (see Fig.15), which contained a number of variables relevant to each tile.

```
//Class to store all data for a single tile
13 references
public class TileData
{
    public float[,] heightMap, heatMap, moistureMap;
    public TerrainType[,] chosenHeightTerrainTypes, chosenHeatTerrainTypes, chosenMoistureTerrainTypes;
    public Biome[,] chosenBiomes;
    public Mesh mesh;

    public Texture2D texture;

    1 reference
    public TileData(float[,] importHeightMap, float[,] importHeatMap, float[,] importMoistureMap,
      TerrainType[,] importChosenHeightTerrainTypes, TerrainType[,] importChosenHeatTerrainTypes, TerrainType[,] importChosenMoistureTerrainTypes,
      Biome[,] importChosenBiomes, Mesh importMesh, Texture2D importTexture)
    {
        this.heightMap = importHeightMap;
        this.heatMap = importHeatMap;
        this.moistureMap = importMoistureMap;
        this.chosenHeightTerrainTypes = importChosenHeightTerrainTypes;
        this.chosenHeatTerrainTypes = importChosenHeatTerrainTypes;
        this.chosenMoistureTerrainTypes = importChosenMoistureTerrainTypes;
        this.chosenBiomes = importChosenBiomes;
        this.mesh = importMesh;
        this.texture = importTexture;
    }
}
```

Fig 15. TileData class with its variables

The GenerateTile function was then altered to become a TileData object, which returned a TileData with all the valid information (see Fig 16.).

```
1 reference
public TileData GenerateTile(float centerVertexZ, float maxDistanceZ, Wave[] heightWaves, Wave[] heatWaves, Wave[] moistureWaves)
{
    Waves Import

    tile depth and width

    x and z offset

    heightmap

    vertex offset and distance between vertices

    heatmap

    moisture map

    build textures

    visualization mode

    update mesh vertices

    //Generate tile data and return
    TileData tileData = new TileData(heightMap, heatMap, moistureMap,
        chosenHeightTerrainTypes, chosenHeatTerrainTypes, chosenMoistureTerrainTypes, chosenBiomes,
        this.meshFilter.mesh, (Texture2D) this.tileRenderer.material.mainTexture);

    return tileData;
}
```

Fig 16. GenerateTile producing tile data

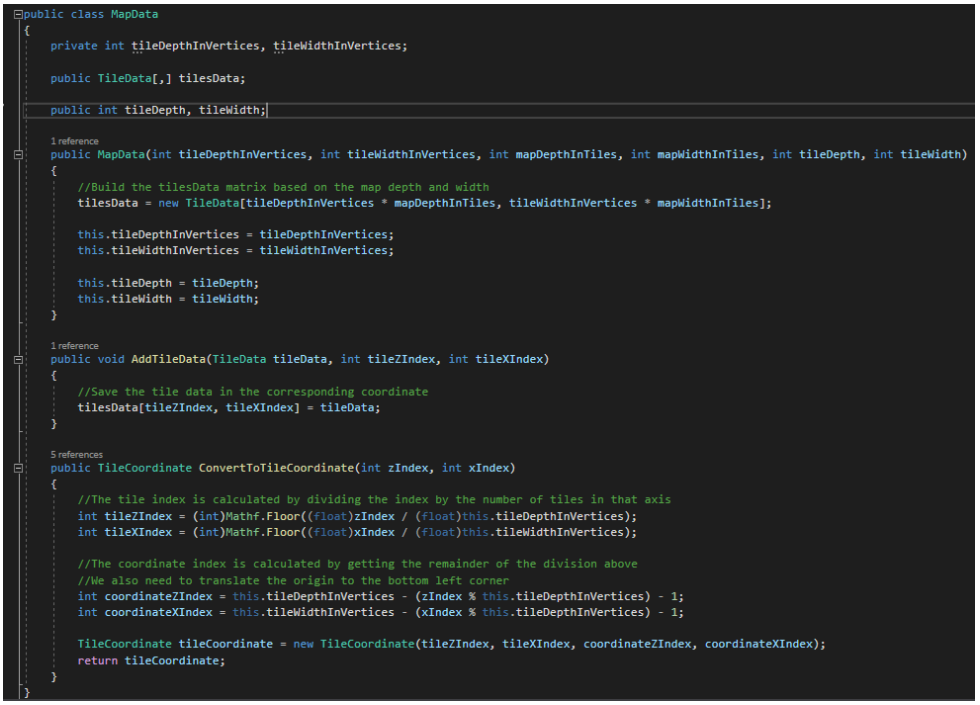
Every tile had its own TileData, and so all TileData objects could be combined into a larger MapData, giving the algorithms the full image of all tiles as one single map. The MapData class created an array of TileData objects based on the width and depth of tiles as well as the number of tiles on the map (as set by the developer). Then, at the generation stage, a MapData object was created and for each generated tile the tile's data was added to the MapData object (see Fig 17.). Finally, due to the tile generation process, each tile used its own local coordinate system for creating height map and biome data. MapData introduced a method for converting coordinates for all tiles into a singular map coordinate system (see Fig 18.).

```
//For each tile, instantiate a tile in the correct position
for (int xTileIndex = 0; xTileIndex < mapWidthInTiles; xTileIndex++)
{
    for (int zTileIndex = 0; zTileIndex < mapDepthInTiles; zTileIndex++)
    {
        //Calculate the tile position based on the X and Z indices
        Vector3 tilePosition = new Vector3(this.gameObject.transform.position.x + xTileIndex * tileWidth,
            this.gameObject.transform.position.y,
            this.gameObject.transform.position.z + zTileIndex * tileDepth);

        //Instantiate a new tile
        GameObject tile = Instantiate(tilePrefab, tilePosition, Quaternion.identity) as GameObject;

        //Generate the tile texture and save it in the map data
        TileData tileData = tile.GetComponent<TileGeneration>().GenerateTile(centerVertexZ, maxDistanceZ, heightWaves, heatWaves, moistureWaves);
        mapData.AddTileData(tileData, zTileIndex, xTileIndex);
    }
}
```

Fig 17. Tile instantiation and map data collection



```
public class MapData
{
    private int tileDepthInVertices, tileWidthInVertices;

    public TileData[,] tilesData;

    public int tileDepth, tileWidth;

    1 reference
    public MapData(int tileDepthInVertices, int tileWidthInVertices, int mapDepthInTiles, int mapWidthInTiles, int tileDepth, int tileWidth)
    {
        //Build the tilesData matrix based on the map depth and width
        tilesData = new TileData[tileDepthInVertices * mapDepthInTiles, tileWidthInVertices * mapWidthInTiles];

        this.tileDepthInVertices = tileDepthInVertices;
        this.tileWidthInVertices = tileWidthInVertices;

        this.tileDepth = tileDepth;
        this.tileWidth = tileWidth;
    }

    1 reference
    public void AddTileData(TileData tileData, int tileZIndex, int tileXIndex)
    {
        //Save the tile data in the corresponding coordinate
        tilesData[tileZIndex, tileXIndex] = tileData;
    }

    5 references
    public TileCoordinate ConvertToTileCoordinate(int zIndex, int xIndex)
    {
        //The tile index is calculated by dividing the index by the number of tiles in that axis
        int tileZIndex = (int)Mathf.Floor((float)zIndex / (float)this.tileDepthInVertices);
        int tileXIndex = (int)Mathf.Floor((float)xIndex / (float)this.tileWidthInVertices);

        //The coordinate index is calculated by getting the remainder of the division above
        //We also need to translate the origin to the bottom left corner
        int coordinateZIndex = this.tileDepthInVertices - (zIndex % this.tileDepthInVertices) - 1;
        int coordinateXIndex = this.tileWidthInVertices - (xIndex % this.tileDepthInVertices) - 1;

        TileCoordinate tileCoordinate = new TileCoordinate(tileZIndex, tileXIndex, coordinateZIndex, coordinateXIndex);
        return tileCoordinate;
    }
}
```

Fig 18. MapData class with tile coordinate conversion

The river generation script could now be created, taking the Map Data as an input variable, thus allowing the algorithm to see the map in its entirety and have rivers flow between tiles. The river generation script had three serialised variables: numberOfRivers, an int; heightThreshold, a float; riverColour, a color. Each of these could be modified in the Unity inspector by a developer to satisfy criteria (1). Simply, river generation was a for loop which runs as long as the desired number of rivers. It called two functions, those are ChooseRiverOrigin, which parsed the map for a random coordinate and checked if it is above the height threshold, selecting it as a valid river spawn if so, and BuildRiver, which was slightly more complicated.

The BuildRiver function created a hash set of Vector 3 coordinates as previous coordinates along the path of the river, thus ensuring the river does not get stuck in a generation loop once it reaches water. A while loop was iterated until water was found. In the loop, the current coordinate was added to the hashset and the algorithm checked if it was a water location. If so, the loop was exited. Otherwise, the coordinate's texture colour was changed to the river colour. After this, the algorithm created a list of all neighbouring coordinates, picked the one with the lowest height value, and continued using this coordinate to build the river until it reached water (see Fig 19.).

With this function in place, the map now generated feasible pseudo-random terrain which was different on every generation, customisable by the developer, and containing multiple distinct biomes as well as rivers. The biome data was also stored within the MapData and TileData objects. This, then, satisfied all criteria laid out for terrain generation.

```
1 reference
private void BuildRiver(int mapDepth, int mapWidth, Vector3 riverOrigin, MapData mapData)
{
    HashSet<Vector3> visitedCoordinates = new HashSet<Vector3>();

    // the first coordinate is the river origin
    Vector3 currentCoordinate = riverOrigin;
    bool foundWater = false;

    while (!foundWater)
    {
        //Convert from map Coordinate System to Tile Coordinate System and retrieve the corresponding TileData
        TileCoordinate tileCoordinate = mapData.ConvertToTileCoordinate((int)currentCoordinate.z, (int)currentCoordinate.x);
        TileData tileData = mapData.tilesData[tileCoordinate.tileZIndex, tileCoordinate.tileXIndex];

        //Save the current coordinate as visited
        visitedCoordinates.Add(currentCoordinate);

        //Check if we have found water
        if (tileData.chosenHeightTerrainTypes[tileCoordinate.coordinateZIndex, tileCoordinate.coordinateXIndex].name == "Water")
        {
            //If we found water, stop
            foundWater = true;
        }
        else
        {
            //Change the texture of the tileData to show a river
            tileData.texture.SetPixel(tileCoordinate.coordinateXIndex, tileCoordinate.coordinateZIndex, this.riverColour);
            tileData.texture.Apply();

            //Pick neighbour coordinates, if they exist
            List<Vector3> neighbours = new List<Vector3>();
            if (currentCoordinate.z > 0)
            {
                neighbours.Add(new Vector3(currentCoordinate.x, 0, currentCoordinate.z - 1));
            }
            if (currentCoordinate.z < mapDepth - 1)
            {
                neighbours.Add(new Vector3(currentCoordinate.x, 0, currentCoordinate.z + 1));
            }
            if (currentCoordinate.x > 0)
            {
                neighbours.Add(new Vector3(currentCoordinate.x - 1, 0, currentCoordinate.z));
            }
            if (currentCoordinate.x < mapWidth - 1)
            {
                neighbours.Add(new Vector3(currentCoordinate.x + 1, 0, currentCoordinate.z));
            }

            //Find the minimum neighbour that has not been visited yet and flow to it
            float minHeight = float.MaxValue;
            Vector3 minNeighbour = new Vector3(0, 0, 0);
            foreach (Vector3 neighbour in neighbours)
            {
                //Convert from map Coordinate System to Tile Coordinate System and retrieve the corresponding TileData
                TileCoordinate neighbourTileCoordinate = mapData.ConvertToTileCoordinate((int)neighbour.z, (int)neighbour.x);
                TileData neighbourTileData = mapData.tilesData[neighbourTileCoordinate.tileZIndex, neighbourTileCoordinate.tileXIndex];

                //If the neighbour is the lowest one and has not been visited yet, save it
                float neighbourHeight = tileData.heightMap[neighbourTileCoordinate.coordinateZIndex, neighbourTileCoordinate.coordinateXIndex];
                if (neighbourHeight < minHeight && !visitedCoordinates.Contains(neighbour))
                {
                    minHeight = neighbourHeight;
                    minNeighbour = neighbour;
                }
            }
            // flow to the lowest neighbour
            currentCoordinate = minNeighbour;
        }
    }
}
```

Fig 19. BuildRiver function

3.2 City Generation and Naming Conventions

3.2.1 Overview

Further to the previously established criteria, the aim of this project was to combine multiple procedural generation methods to create a more believable, detailed map generation system. To this end, points of interest were to be instantiated into the map, in this case exclusively cities to simply prove the viability of the system. The system, then, should (6) generate a series of cities in viable,

believable locations; (7) make use of the terrain generation system to improve the believability and viability of the cities.

To these ends, the generator would seek to instantiate cities on land instead of in water (so that they have a viable grounding as real cities do) (6), as well as ensuring cities do not appear too close to each other. Additionally, it would name cities according to the biome that they appear in, using biome data from the terrain generation, thus cities near each other and in the same biome should have had similar naming conventions, and so appear to have risen from the same cultural origins (7).

3.2.2 City Instantiation

In order to instantiate cities, a city prefab was created and saved, similarly to the tile prefab. This prefab simply had a cylinder object and a text object, thus allowing it to display a location and a name (see Fig 20.).

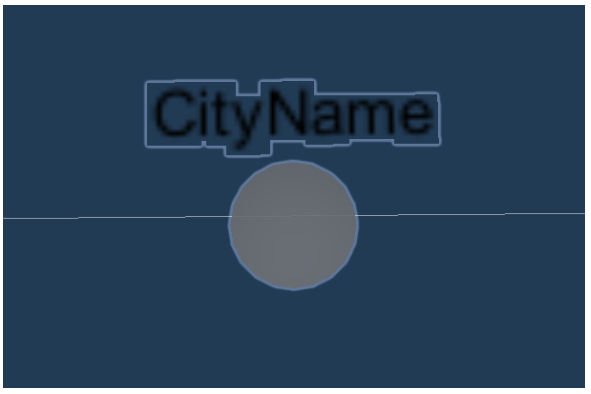


Fig 20. City Prefab within the prefab inspector

Then, the CityGeneration script was created, and contained serialised variables numberOfCities, an int; spawnRadius, an int; and cityPrefab, a GameObject. It also had an empty list of GameObjects called cities, which would store all instantiated cities. This became important as it was necessary to ensure cities don't spawn too closely to each other, and that the developer can determine for themselves how many cities should spawn on the map, thus satisfying criteria (1), (2) and (6).

The GenerateCities function within the script ran a for loop as long as the number of cities the developer had input. In each loop, the ChooseCitySpawn function was called, with map depth and width as well as map data as input variables, then it instantiated a city in the chosen spawn point. It then accessed the text of the city prefab and generated a name based on the biome the city was in using map data and the city's spawn point. Finally, the city was added to the list of cities.

The ChooseCitySpawn method worked similarly to the ChooseRiverOrigin method, except for some qualifiers for the spawn point. It first chose a random point on the map, converted that to tile coordinates, and then checked the biome of that coordinate as opposed to a height threshold. If the biome was not null, it meant it was not water, and so it was on land and therefore viable. Then, the system checked for cities within the developer's input spawnRadius, which determined how close one city can be to another. It did so by using a foreach loop, iterating through every member of the cities list and checking the distance between the city's spawn and this potential spawn point. If there were no cities in this distance, the spawn point was considered found and the algorithm returned the point as a viable spawn (see Fig 21.).

```
1 reference
public Vector3 ChooseCitySpawn(int mapDepth, int mapWidth, MapData mapData)
{
    bool found = false;
    int randomZIndex = 0;
    int randomXIndex = 0;

    //Iterate until find a good spawn point
    while (!found)
    {
        //Pick a random coordinate inside the map
        randomZIndex = Random.Range(0, mapDepth - (mapDepth / 5));
        randomXIndex = Random.Range(0, mapWidth - (mapWidth / 5));

        //Convert from map coordinate system to tile coordinate system and retrieve corresponding tile data
        TileCoordinate tileCoordinate = mapData.ConvertToTileCoordinate(randomZIndex, randomXIndex);
        TileData tileData = mapData.tilesData[tileCoordinate.tileZIndex, tileCoordinate.tileXIndex];

        //Ensure cities can't spawn in water (meaning they can't spawn in areas without a biome)
        if (tileData.chosenBiomes[tileCoordinate.tileZIndex, tileCoordinate.tileXIndex] != null)
        {
            Biome checkBiome = tileData.chosenBiomes[tileCoordinate.tileZIndex, tileCoordinate.tileXIndex];

            Vector3 checkPoint = new Vector3(randomXIndex, 5, randomZIndex);

            //Check radius for nearby cities
            if (cities.Count > 0)
            {
                foreach(GameObject city in cities)
                {
                    if (Mathf.Abs(Vector3.Distance(checkPoint, city.transform.position)) >= spawnRadius)
                    {
                        found = true;
                    }

                    else
                    {
                        found = false;
                        break;
                    }
                }
            }

            else
            {
                found = true;
            }
        }
    }

    return new Vector3(randomXIndex, 5, randomZIndex); //We use 5 as the y coordinate as the map is only to be viewed top-down, and so the perceived elevation is irrelevant. It just has to be above the map terrain.
}
```

Fig 21. ChooseCitySpawn

With this, the city was instantiated in a viable location, determined by developer input, with stored information about the biome it is located in, thus satisfying criteria (1) and (6) (see Fig 22.).

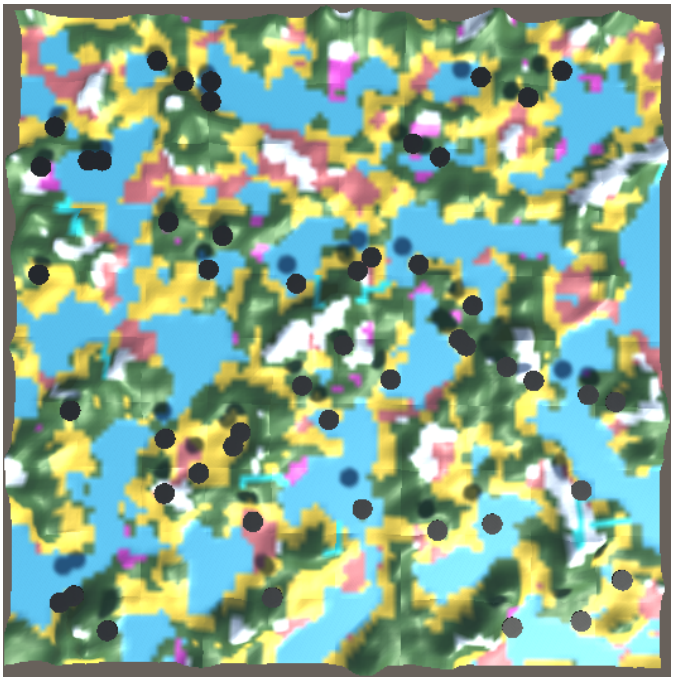


Fig 22. Cities spawning away from water. Note minimal separation here, which can be altered.

3.2.3 City Naming

Once a city was instantiated, it was given a name based on the biome it was in. To do this, the GenerateCityName function was called, with the city spawn location and map data as input. Firstly, the function converted the spawn location to the tile coordinates, then used the map data to check the biome of this coordinate. The result of this search was passed through a switch, and the output

for the name was dependent on the result (see Fig 23.). The way this function operated, a series of prefixes, affixes (labelled as “middleWord”), and suffixes were manually generated. Each biome had a unique list of prefixes and suffixes, with the affixes being shared between them all (see Fig 24.). When a biome was identified, the algorithm accessed the list of assigned prefixes and randomly selected one, then did the same for suffixes. In all cases, the algorithm then randomly selected an affix. These three components were then simply added together to form a single string, and that string was returned and applied to the text component of the city prefab.

For the purposes of this project, given the scope, the selected prefixes and suffixes were limited in variability. This was to give a clear sense of lineage to each biome’s naming conventions, which satisfied criteria (7).

With this process complete, the map generator spawned multiple tiles upon which a Perlin noise was randomised and applied to create height, heat and moisture maps. These were used to determine the landscape of the terrain as well as create unique biomes across the map. This data was stored and then used to generate cities which are named based on the biome in which they resided, creating groupings of cities from the same apparent cultural lineage (see Fig 25.). This, then, satisfied all criteria.

```
switch (cityBiome.name) //Check the biome name for the coordinate, then generate the prefix and suffix accordingly
{
    case "Savanna":
        cityPrefix = savannaPrefix[Random.Range(0, savannaPrefix.Length)];
        citySuffix = savannaSuffix[Random.Range(0, savannaSuffix.Length)];
        break;

    case "Desert":
        cityPrefix = desertPrefix[Random.Range(0, desertPrefix.Length)];
        citySuffix = desertSuffix[Random.Range(0, desertSuffix.Length)];
        break;

    case "Grassland":
        cityPrefix = grasslandPrefix[Random.Range(0, grasslandPrefix.Length)];
        citySuffix = grasslandSuffix[Random.Range(0, grasslandSuffix.Length)];
        break;

    case "Tundra":
        cityPrefix = tundraPrefix[Random.Range(0, tundraPrefix.Length)];
        citySuffix = tundraSuffix[Random.Range(0, tundraSuffix.Length)];
        break;

    case "Boreal Forest":
        cityPrefix = borealPrefix[Random.Range(0, borealPrefix.Length)];
        citySuffix = borealSuffix[Random.Range(0, borealSuffix.Length)];
        break;

    case "Tropical Rainforest":
        cityPrefix = rainPrefix[Random.Range(0, rainPrefix.Length)];
        citySuffix = rainSuffix[Random.Range(0, rainSuffix.Length)];
        break;
}

cityMiddle = middleWord[Random.Range(0, middleWord.Length)];

string cityName = cityPrefix + cityMiddle + citySuffix;

return cityName;
```

Fig 23. Switch for determining the biome of the city spawn and naming accordingly

```
//Arrays for prefixes, middles, and suffixes of city names
string[] savannaPrefix = new string[] { "Sa", "Sav", "San", "Su", "Za", "Zu" };
string[] grasslandPrefix = new string[] { "Al", "An", "Ad" };
string[] desertPrefix = new string[] { "Dak", "Das", "Dol" };
string[] tundraPrefix = new string[] { "Ske", "Skja", "Sek" };
string[] borealPrefix = new string[] { "Tor", "Tav", "Tul", "Tol" };
string[] rainPrefix = new string[] { "Bra", "Ban", "Bol" };

string[] middleWord = new string[] { "an", "ad", "va", "dar", "da", "la", "len", "liv", "ver", "vil", "bad", "cad", "dav" };

string[] savannaSuffix = new string[] { "a", "ah", "o", "ya", "da" };
string[] grasslandSuffix = new string[] { "ren", "id", "ds" };
string[] desertSuffix = new string[] { "ba", "ca", "vo" };
string[] tundraSuffix = new string[] { "a", "e", "oe" };
string[] borealSuffix = new string[] { "on", "en", "ak", "adh" };
string[] rainSuffix = new string[] { "go", "ga" };
```

Fig 24. List of all prefixes, affixes and suffixes according to biome where appropriate

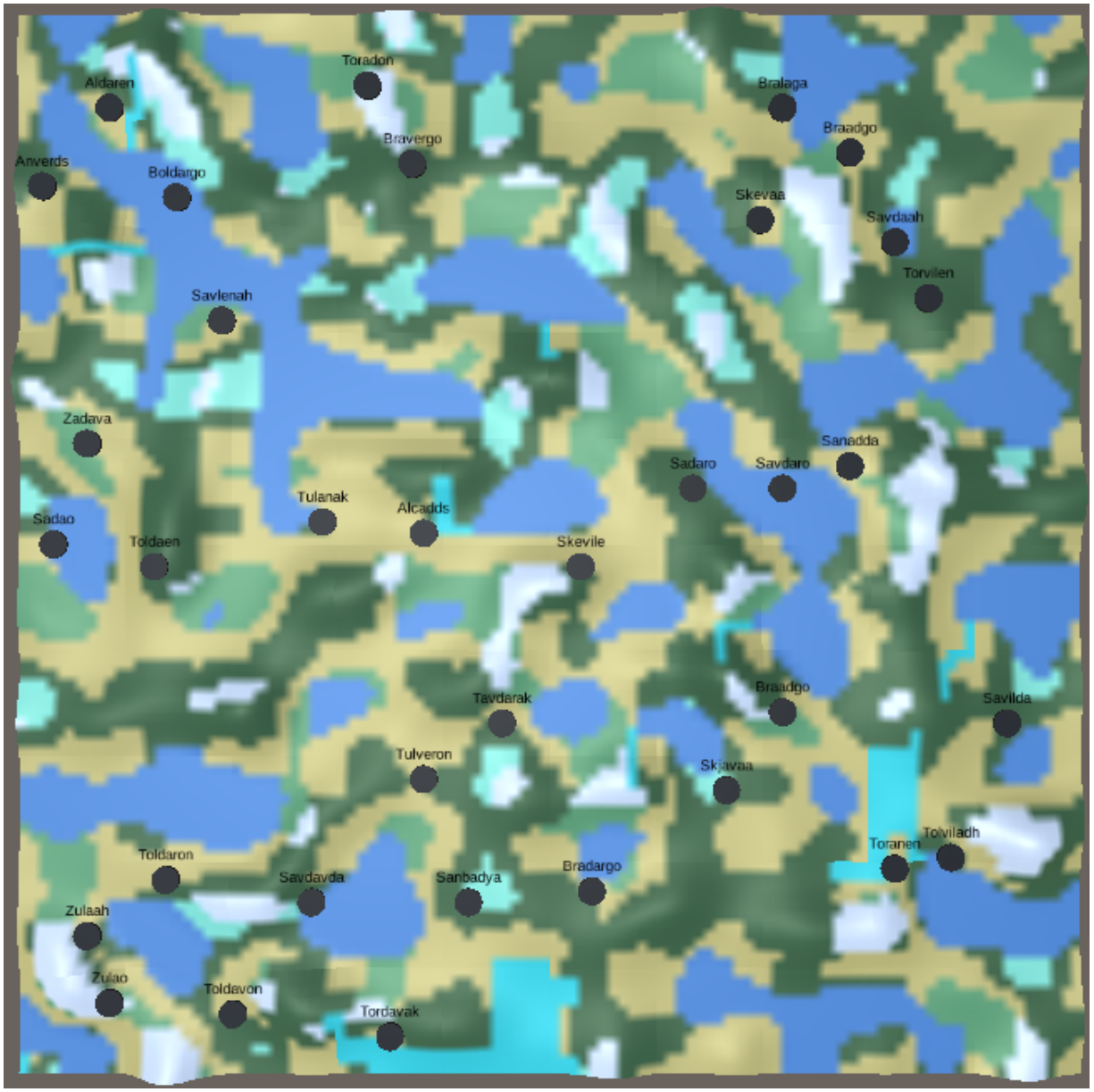


Fig 25. A map generated with city separation and names based on biome

3.3 Survey and Data Collection

3.3.1 Overview

To test the viability of the map generation system, it was determined that user feedback and data were required. Maps are designed and made for users to engage with, so it is important that they find the system matches the criteria outlined at each stage of execution. Criteria (1) and (4) were exclusively backend issues, with (1) touching on developer needs and (4) being a stage in the generation which allowed the satisfaction of (6) and (7). Therefore, (1) and (4) did not require user testing, and so the data collection process would assess the success of the generator based on the remaining criteria.

Due to safety measures regarding the Coronavirus Pandemic at the time the project was undertaken, it was vital to create an online solution for the data collection process, since it would prove extremely difficult to organise having a larger quantity of respondents gather together at one time to complete it. Based on this fact, the final build of the project was created as a WebGL program and uploaded to a hosting site to be accessed remotely. Then the decision was made to conduct the research as an

online survey. This method allowed many questions to be answered efficiently and without taking up too much of the respondents' time, thus incentivising participation. Additionally, it complimented the online format well, and using Google Forms allowed for automatic generation of valuable data comparisons, and the ability to view feedback on a population or individual basis, and as a whole or by question.

3.3.2 Map Regeneration

In order to test criteria (3), which states that the map generator must create a unique map on every iteration, the final build of the map generator added a "Regenerate Map" button. This was tied to the ReloadScene script (see Fig 26.) and simply used Unity's built in scene management to load the map generation scene again. This, then, reloaded and ran all code again, including wave randomisation, city spawning and naming, and so the map was unique on every generation.

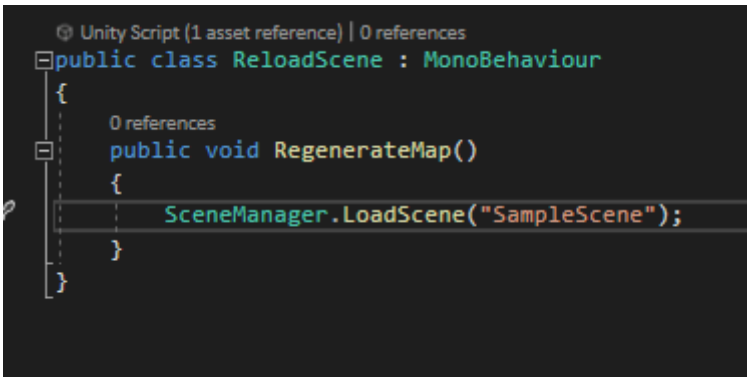


Fig 26. Reload Scene function ties to the button

3.3.3 Survey Construction and Flow

As part of the conduction of this survey, respondents were not necessarily expected to have a strong familiarity with map generation software or maps themselves. This allowed for a distinction to be made between users who did and did not have that familiarity, and a subsequent analysis of how respondents' knowledge of map generation impacted the perceived believability of the map. Therefore, the survey was targeted at any adult who could legally consent to participation.

To first understand the level of investment respondents would have in the software, and to make the distinction between those who did and did not have experience with such software, respondents were asked about their familiarity with world maps in games, as well as more specific familiarity with similar map generation tools. To allow for a range of familiarity and interest, as these are not typically static or consistent across respondents, a scale of 1-10 was utilised, using a "forced choice" Likert scale. This stops respondents from choosing a "middle" response, and has the data more obviously distributed into two distinct groups of familiarity, whilst also preserving some nuance for more detailed analysis (Allen, I.E and Seaman, C.A., 2007). The forced choice Likert scale was used throughout the survey for the same reasons.

Criteria (2) was that the map is feasible. To test this with respondents, they were asked how believable they found the layout of the map, ranging between "not believable at all" and "very believable". This provides a solid grounds for comparison between multiple map iterations, and again allows for grouping responses into overall agreement or disagreement with whether the map was believable.

Criteria (6) was to generate cities in viable, believable locations. This criteria is closely aligned and related to criteria (2), and therefore acted as the subject for the follow-up to question 1. Respondents were asked how well spread the city markers were across the map, ranging from “very bunched together” to “well spread out”. This tests the visual appearance of the distribution as well as whether the city generation method is properly accounting for distance between cities.

Criteria (4) was to produce distinct biomes, which also relates to the immediate visual satisfaction of the map as per criteria (2) and (6), and so was the subject of the next question. These first questions were in service of the immediate visual response, somewhat ignoring the finer details and capturing that initial visual feedback. The response here was a multiple choice, where respondents were asked if they noticed distinct biomes (with a brief explanation on what a biome was in case they were not familiar with the term). Responses were “no”, “one or two” or “several”. This was to examine whether the biome system was working at all, and if so working to the desired level of terrain variation as per criteria (4)’s use of a Whittaker diagram.

At this point in the survey, the respondent’s immediate initial visual feedback should be captured, and so later questions were designed to hone in on finer details, specifically further analysis of criteria (6) and (7) as a means of fulfilling and improving the satisfaction of (2). This stage of analysis effectively determines the viability of the project as an answer to the research question, as responses detailed whether the biome and city generation system were working together effectively.

The first of this series of questions asked about the diversity of generated names. The responses were expected to be roughly in line with responses to the previous question about the variety of biomes, since the naming scheme uses biomes to determine names. This question used a scale in order to gauge a quality of diversity, ranging between “lots of repetition” and “very diverse”. This is specifically referring to criteria (7).

The following questions again sought to verify criteria (7). It was important, and one of the key reasons for developing the project, that the cities be seemingly grouped into distinct cultural regions. To this end, the first question asked if there were any noticeable groupings to cities. This would either immediately register to respondents or they would be able to reassess their generated map for this detail. This question, and the following, used a scale also, so as to identify specifically how effectively cities seemed to be showing patterns through the mixture of biome data, spawning radii and avoiding water. The following question asked respondents whether cities could be grouped into recognisable “countries”, or areas where naming conventions appeared to be similar. This specifically targets the names within any potential groups of cities, and the comparison between these two responses should provide insight into the viability of the tool.

Respondents then were asked to regenerate the map and answer the same series of questions regarding the new map. In order to test criteria (3), that the software generates a new map on every iteration, as well as test each map against the other criteria, respondents were asked a series of questions on one map, then asked to regenerate a map and answer the same questions on the second. This sequence should highlight any obvious similarity in content between the two maps the user generates, and therefore test the viability and diversity of maps across multiple generations. Respondents would then be asked a series of questions regarding the differences between the maps, both attesting to the quality of each and the uniqueness, again testing criteria (3) and testing all maps against other criteria also. Asking respondents to highlight the perceived differences between maps was also a method to have them think critically of the individual aspects, and therefore provide qualitative and detailed feedback regarding possible improvements to the system. To this end, the comparisons between maps contained both quantitative Likert scale responses, as well as qualitative

written ones. Having respondents each generate two maps was designed to provide insight into direct comparisons between maps, but also increased the data pool for the number of maps being assessed, doubling it. More maps being assessed for viability increases the effectiveness of the evaluation method.

The first question of this final section was designed to make the respondent think about the differences between the two maps. It asked respondents to select one map as the more believable world. This has them think critically about the differences between the maps, using their previous analysis to come to a conclusion of the overall effectiveness and viability of the generated world as per criteria (2) and (3). The following question asked for specific, written detail on their thought process behind the decision. In these answers, it was expected that respondents would give valuable information regarding potential shortcomings in the tool over multiple map generations.

The following question asked users if they noticed naming themes across both maps, and if they found them interesting, asking them to further explain their answer. This gives some qualitative, written feedback on the naming system between map generations for further analysis on potential adjustments to be made to the system, and a more detailed feedback for criteria (7).

The final question in the survey asked respondents whether they overall felt the tool was an effective one, and whether the produced maps were of a good enough quality to be used. This was an effective way to sum up the respondent's opinions on the tool as a whole, having seen multiple map generations, thought in detail about each map, and having a better understanding for the purpose of the tool. This question, ultimately, asks users whether maps are generated effectively, assessing all criteria.

4. Evaluation

4.1 Respondents Analysis

Over the course of the testing period, the survey received 23 total responses. Of the respondents, 21 out of 23 expressed strong familiarity with overworld maps, 16 of whom responded with a 10, the maximum available response. Additionally, 12 of the 23 respondents claimed to have used other map generation tools, and 16 of 23 suggested interest in map generation, with 14 respondents responding with an 8 or higher.

From this data, it could be gathered that the respondent pool was largely familiar with overworld maps, which was something well suited to an analysis of the quality of maps produced. This isn't to say that the two respondents less familiar with maps created unusable or unhelpful data - it was still important to analyse the effectiveness of the map generation tool on a broader audience. Unfortunately, this would not qualify as a significant enough respondent pool to justify making those comparisons. However, almost exactly half of respondents claimed to have used other tools, which allowed for an excellent comparison between users who have and have not used others, acting as a means to compare the viability of this solution against others.

4.2 Terrain Generation and City Spread Results

In regards to the first question of the believability of the map layout, across both maps, respondents voted in the majority to favour the maps as believable, with 24 responses as 6 or above and 22 as 5 or below (46 total maps were generated for the research, based on two maps per respondent). This is

a narrow majority, with approximately 52% of responses claiming the map to be overall believable. It should be noted, however, that the lower responses were weighted towards believable, with no responses in 0, and 17 of the 22 negative responses between a score of 4 and 5.

Further breaking down the results between respondents who were familiar with map generation tools or not yielded additional insight. Respondents familiar with other tools tended not to believe the maps were believable, with a mean response of 5.5 and a mode of 4 (which had 8 of the 22 responses). Respondents unfamiliar with other tools tended to favour the maps as believable, with a mean response of 6.3 and, interestingly, a mode of both 5 and 8 (which each received 5 responses of the 24). For those unfamiliar with other tools, 14 of the 24 responses were 6 or above, and for those familiar 9 of the 22 were 6 or above.

Overall then, respondents who had used other tools were less convinced of the viability of generated maps, despite the overall positive result and particularly positive result amongst respondents who had not used other tools. This, then, could possibly be attributed to an increased scepticism amongst respondents with other experiences, or show signs that other solutions have provided more feasible maps. Generally, though, in the case of all respondents and both subgroups, the response is almost equally divided, and so the result is undetermined as to criteria (2). Results were additionally consistent across both maps, showing that both maps, at the very least, were equally viable to each other, qualifying criteria (3).

For city spread, results appeared more favourable. 31 of 46 responses were 6 or above, thus favouring the generator in terms of city spread. That equates to 67.4% of responses being favourable ones, with a mean result of 6.76 and a mode of 8. This shows generally that cities were appropriately spread out in the majority of generated maps, satisfying criteria (6). For respondents who had previous knowledge of other map generation tools, 13 of 22 responses were 6 or above (59%), with a mean result of 6.86.

It seems, then, that the tile/perlin-based terrain generation system itself was undetermined as to whether it produced a viable, believable map as per criteria (2), but the algorithm for spreading cities whilst avoiding water and other cities in a radius may be a reasonable solution to criteria (6).

4.3 Naming and Grouping/Biome Recognition Results

When asked whether they noticed distinctive biomes in the map, the strong majority of respondents reported seeing several biomes present. Of the 46 generated maps, in 34 of them several biomes were noticed, with only one or two noticed in 12 maps. No responses reported a complete lack of distinct biomes. So 73.9% of responses observed multiple biomes, strongly suggesting success in distributing biomes between maps, satisfying criteria (4).

When asked about diversity of city names, 25 of the 46 responses were 6 or above (54.3%), with a mean result of 5.87 and a mode of 7. The results were slightly more favoured by those without prior knowledge of map generation systems, however the difference is somewhat negligible, with respondents without prior knowledge producing a mean result of 5.92 and a mode of 5, against those with prior knowledge producing a mean result of 5.82 and a mode of 7. Responses of 6, 7 or 8 made up 10 of the 22 responses for respondents with prior map generation experience, suggesting a largely favourable result. In the same subsect of respondents, 13 of the 22 responses were in the favourable half, 6 or above, which again shows promising results at 59.9%.

When asked if they noticed any distinct patterns or groupings to cities, respondents overall responded in the positive, though only marginally, with 25 of 46 responses scoring 6 or higher (54.4%), a mean score of 5.61 and a mode of 8 (10 responses). However, this response had the largest deviation between respondents with and without knowledge of other map generation systems. For respondents who had used other systems, 18/22 responses were 5 or lower (81.82%), with a mean response of 4.05 and a mode of 5. For respondents without knowledge of other tools, responses were significantly more favourable, with 21/24 responses being 6 or above (87.5%), a mean score of 7.04 and a mode of 8.

When questioned if any recognisable countries or areas where naming conventions were similar emerged, respondents were divided, with exactly 50% of responses each between the lower 1-5 and higher 6-10 tiers. The mean response was 5.65, a favourable result, and the mode was split evenly between 4 and 5, with 7 responses each. Again, this question produced a larger disparity between those with and without prior map generation tool experience. Respondents with prior experience reported a mean response of 4.05 and a mode of 5, with 18/22 responses being 5 or lower (81.82%). Meanwhile, those without prior knowledge of map generation tools reported a mean score of 6.46, a significantly higher result, and a mode of 8, with 17/24 responses being 6 or above (70.83%).

In summary, then, results show a trend towards the map generation system producing noticeable grouping to cities and appropriate names, seemingly giving the appearance of shared heritage. However, this result is rather marginal, so although showing some positive signs the system may require refinement in order to satisfy criteria (7). It was noted that there was a significant disparity in these responses between those who had and had not used other map generation tools in the past, which may have implied that the solution in its current state was worse than existing alternatives, but this did not disqualify the evident potential of the solution. On a more positive note, the solution definitively satisfied criteria (4), as respondents overwhelmingly believed the system produced several distinct biomes.

4.4 Map Comparisons

4.4.1 Purpose

At this stage, respondents were asked to choose between maps 1 and 2 as to which made for a more believable world. This question was designed to have the respondents think critically about specific differences between the two maps, thus allowing them to produce more detailed, qualitative responses as to different map iterations in the following questions. It was hoped that this data could shed insight on previously answered questions and any resultant discrepancies.

Unfortunately, this question was somewhat flawed in its approach. Providing a third option, to determine that both maps were about equal in their believability, would have aided in analysing the generator’s satisfaction of criteria (2) and (3), in which it should produce unique, feasible maps on each iteration. At this stage, forcing users to choose between the maps did have them think critically on the differences, but failed to account for the possibility that both maps performed well.

Regardless, the result was somewhat interesting, with 13 respondents of 23 (56.5%) favouring the second map. What made this result interesting is that, in almost every question, the second map received less favourable responses. This is an interesting contradiction, as it suggested respondents were more critical of the second map, but also preferred it in the majority. This may be due to the formation of the survey - respondents had no frame of reference for the first map beyond world maps from pre-existing sources, whether those be games or other map generators. For the second

map, however, users were directly (if not intentionally) comparing with the first map, with a foreknowledge of the details they would be questioned on and therefore have to look for. This may have accounted for the seeming contradiction in these results, but this was difficult to definitively prove.

4.4.2 Difference Between Map Iterations

When asked to comment on why they chose one map over the other, a consistent theme emerged. Respondents claimed that one map spread cities in a more believable way than the other. This initially led to a belief that the city spawning algorithm was flawed, and two respondents commented that cities appeared just inside bodies of water, but this conclusion contradicted previous findings, in which respondents strongly agreed that cities were well spread across maps. So although the city spawning algorithm may require slight refinement (likely due to the animation curve creating an artificial barrier between the coast and water, thus making cities on the coast appear to be in the water), particularly at the boundaries of biomes (where multiple cities could spawn in relatively close proximity with different naming conventions, due to both appearing on the edge of their respective biomes in areas where more than two biomes met), the overall dispersion of cities seems to be fine. In fact, several comments noted the believable dispersal of cities. Other comments perhaps justify this explanation - several respondents commented that names appeared too similar in close proximity to each other.

4.4.3 Comments on Naming Conventions

Due to the scope of the project and the time required to deeply analyse etymological origins of words to form multiple languages for city names, it was decided to simplify the naming process and make it scalable so that future developers could add complexity to the naming schemes and thus add complexity to the maps as a whole. In the question regarding the naming conventions, all respondents noted that they noticed and mostly appreciated the naming themes, however several respondents also mentioned regularly repeating names or syllables, as well as a regular repeating of the first letter. This was created as the signifier for each culture (see Fig 24.) to easily distinguish each culture from the next, but the lack of variety may have actually cheapened the effect and made respondents view it negatively, thus affecting their final opinions on city spawns (despite an initial favour towards city dispersal).

Overall, then, the naming conventions of each culture seemed to be at fault for several issues with the solution, but these in fact successfully satisfied criteria (7) - respondents were able to identify patterns in naming convention and recognised the pattern as being by biome. In fact, several respondents noted that they had thought the cities were grouped into countries, not biomes, implying a successful illusion of emergent culture in the map.

A potential flaw with this line of questioning was that it specifically asked for negative feedback, tasking respondents with explaining why one map was better than the other, and by inference why one map was worse. It could also have asked what the strong aspects of the generator were, as opposed to only highlighting the perceived weaker elements (although this method has produced a significant finding).

4.5 Overall Result

When asked if overall, did they think the map generation tool was effective, respondents overwhelmingly responded yes (73.9%) as opposed to no (26.1%). For respondents with knowledge

of other tools, this result was still positive, with 54.54% believing it an effective tool. 91.6% of respondents with no prior map generation tool familiarity believed the tool to be an effective one.

Generally, the solution was positively received across the board. Although it was undetermined whether it satisfied criteria (2), it showed promise in satisfying criteria (7), and definitive results in satisfying criteria (3), (4), and (6).

5. Conclusions and Future Work

5.1 Overview

To conclude, the project set out to utilise multiple facets of content generation to create a more complex, interesting piece of content, as opposed to simply focusing on individual pieces of generated content, and as a result produce a world map with believable naming conventions for points of interest in distinct biomes. To this end, and in conjunction with literature, seven criteria were outlined; the solution should (1) enable developers to alter variables to change the map to their purposes; (2) generate feasible terrain; (3) produce a unique map on every iteration; (4) produce distinct biomes; (5) store data for biomes at certain coordinates; (6) generate a series of cities in viable, believable locations; and (7) make use of the terrain generation system to improve the believability and viability of the cities and map at large.

Based on internal and user testing, the solution proved to meet all criteria, although requires future refinement to accentuate these successes. The system allowed several controls for developers to alter the appearance of the terrain, biomes, cities and city names. It generally produced feasible terrain, although alterations to the animation curve determining the appearance of water could be made, and the height threshold of water to reduce the overall abundance in the maps. The system did provide unique maps on every iteration, each with distinct biomes. The system did store biome data through the tile data object and map data class, and this was used in conjunction with the successful city spawning mechanic to name cities according to the biome they spawned in, thus improving the believability of the map at large and creating the illusion of emergent cultures.

5.2 Future Work

To further iterate on this tool, several systems could provide the refinement required to greatly improve its effectiveness.

Firstly, one could look more deeply into etymology and the way that words from different cultures are formed, incorporating this research into the biome-based naming solution. This would increase the diversity of names across the generated maps and provide much more believable, detailed naming conventions whilst still maintaining the evident cultural groupings. Further to this, one could create multiple possible cultures for each biome, using the city spawn radius function and expanding upon it to ensure that only cities within a local area and the same type of biome had one unique culture. This would further still diversify the names in the map, and likely lead to a true emergence of noticeable countries (as opposed to multiple names from the same culture appearing globally, they would instead be more localised and could be “claimed” and removed from the available culture list after instantiation).

An additional potential iteration on the solution would be to have other points of interest (POIs) spawn within a radius of each city, using a system similar to the city spawn radius. This could allow

other POIs like towns, forests, dungeons, caves and more to be spawned based on the culture of the local city, as well as other factors like the height of the POI.

For the generation of terrain, the tiling method was successful in that respondents did not notice the gaps between tiles. However, the landscape was the most critiqued element of the solution. One possible solution is to add lacunarity - a smoothing effect which may reduce the irregularity of the terrain. Another possible solution is the use of simplex noise instead of Perlin noise. Simplex noise is a recently emerging noise solution which shows great promise as an alternative to Perlin for the purposes of generating maps, and may prove to be a viable alternative when used in conjunction with a biome system.

Appendix

Survey Responses

Survey Intro/Description

Procedural Map Generation Survey

The purpose of this survey is to assess the effectiveness of a world map generation tool, which creates a unique map every time you generate. "Overworld" maps, as generated here, are often used in fantasy and adventure settings to show the layout of the fictional world. These maps are used by game writers, authors and tabletop roleplaying games fans to facilitate exploration and sense of space in the world.

This survey should take around 10 minutes to complete. In order to complete the survey, you need to be on a desktop PC and access the following website:

https://simmer.io/@SwanDiveGames/honours-random-world-generation-w-cities-named-by-region

This is a safe site, and hosts the program online. Once you're on the webpage, simply follow the survey through. Thank you for your time.

Please note that no personal information is kept for the purposes of this research. All information will be anonymised. By completing this survey, you agree that your answers to the questions may be analysed and used for the purposes of the research project.

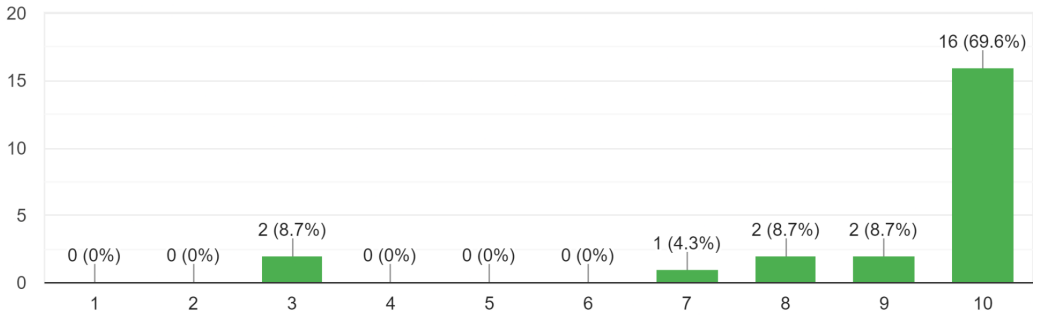
If you encounter any issues, please contact markflan34@outlook.com for help.

Respondent Analysis

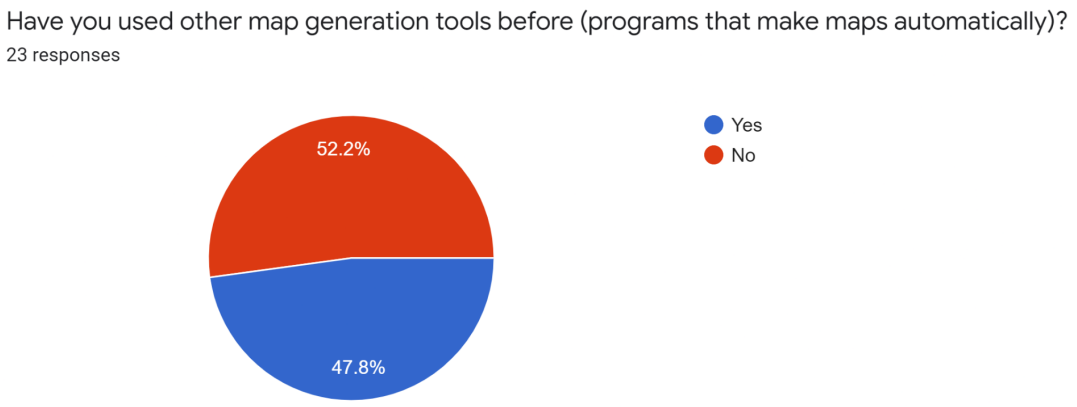
Question 1

How familiar are you with "overworld maps", whether from video games like Final Fantasy and Elden Ring or from tabletop games like Dungeons & Dragons and Warhammer Fantasy?

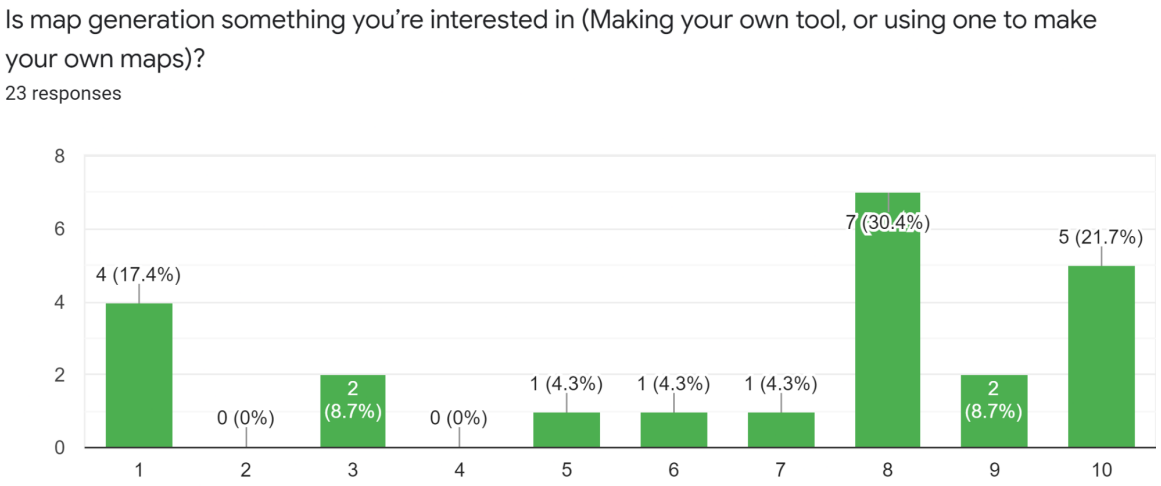
23 responses



Question 2

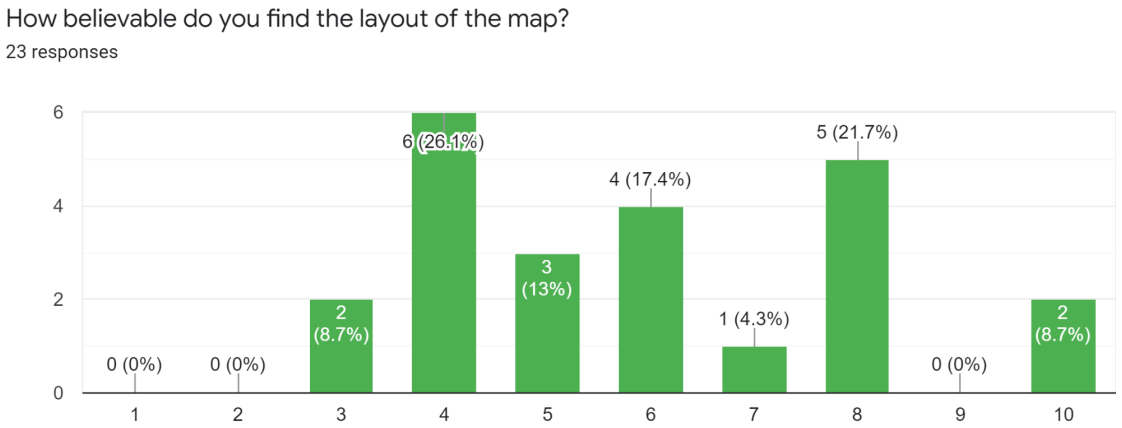


Question 3

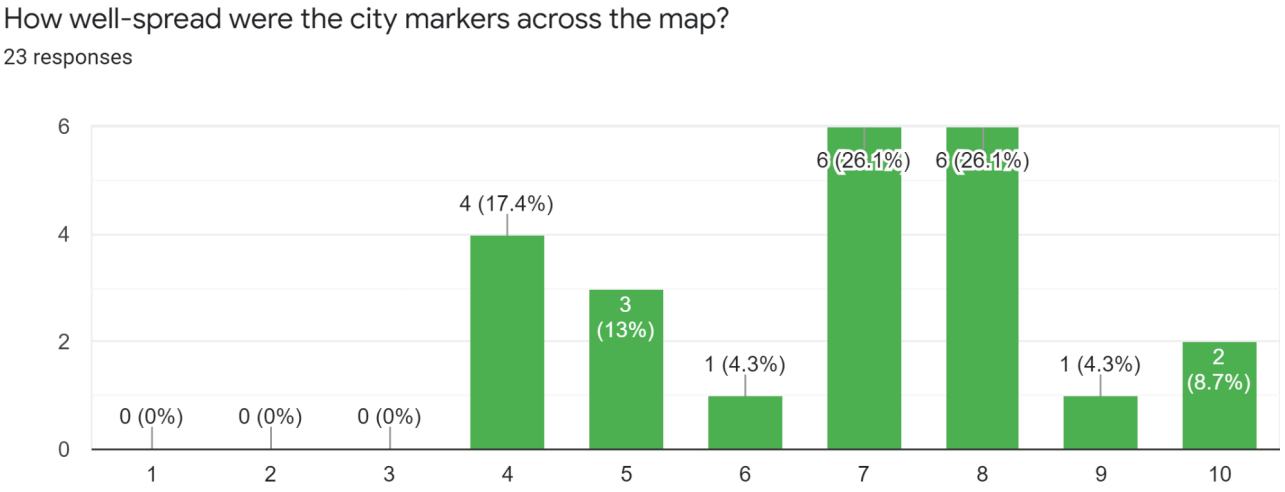


Map 1

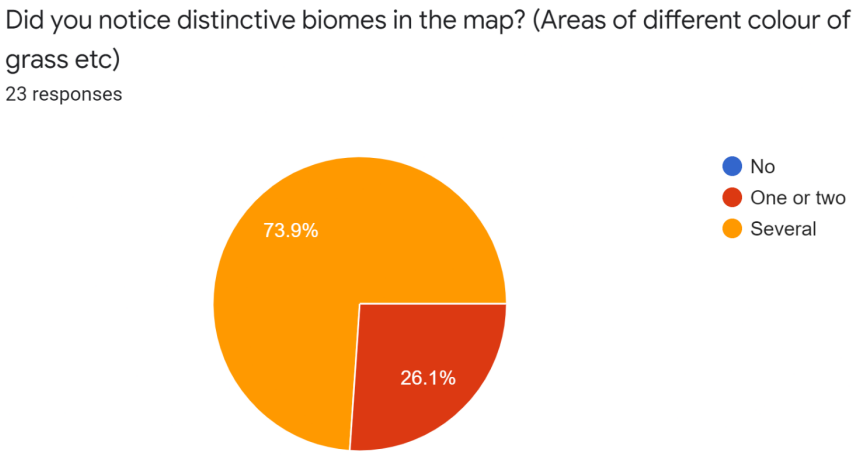
Question 1



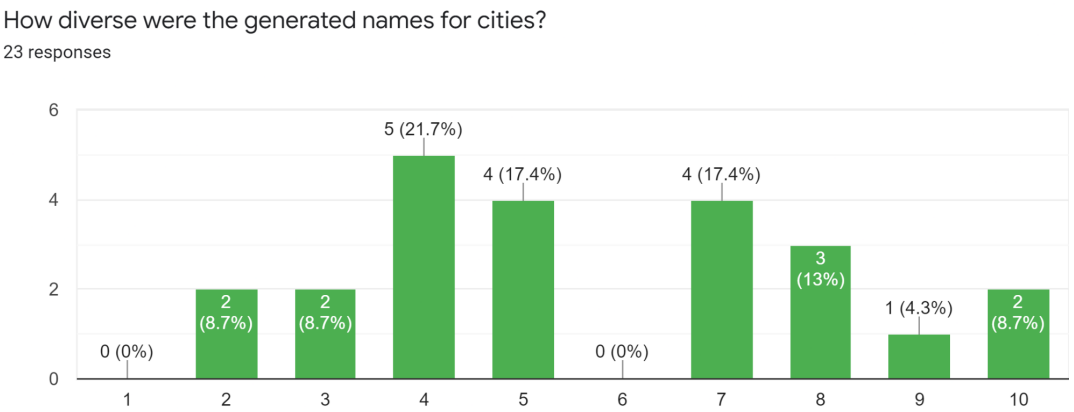
Question 2



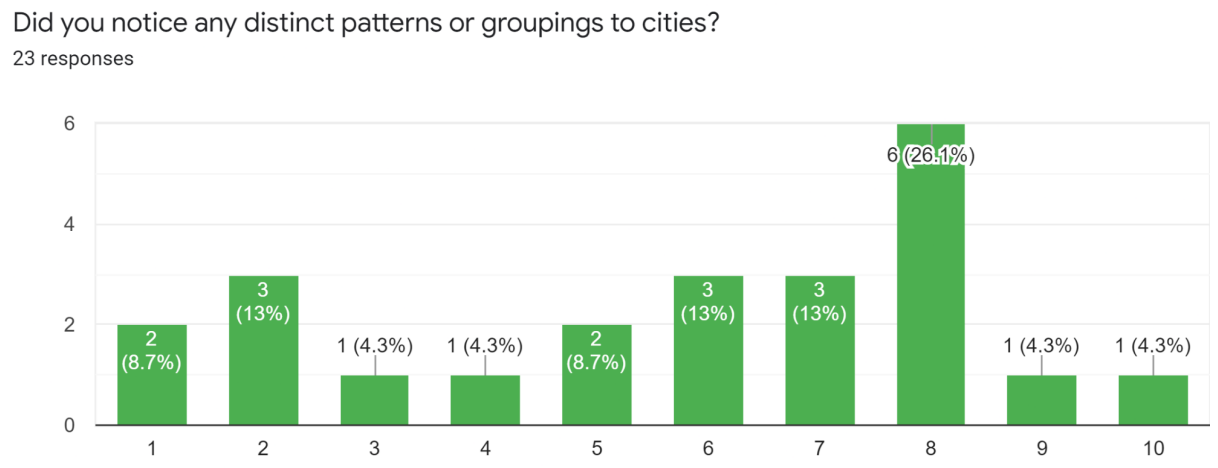
Question 3



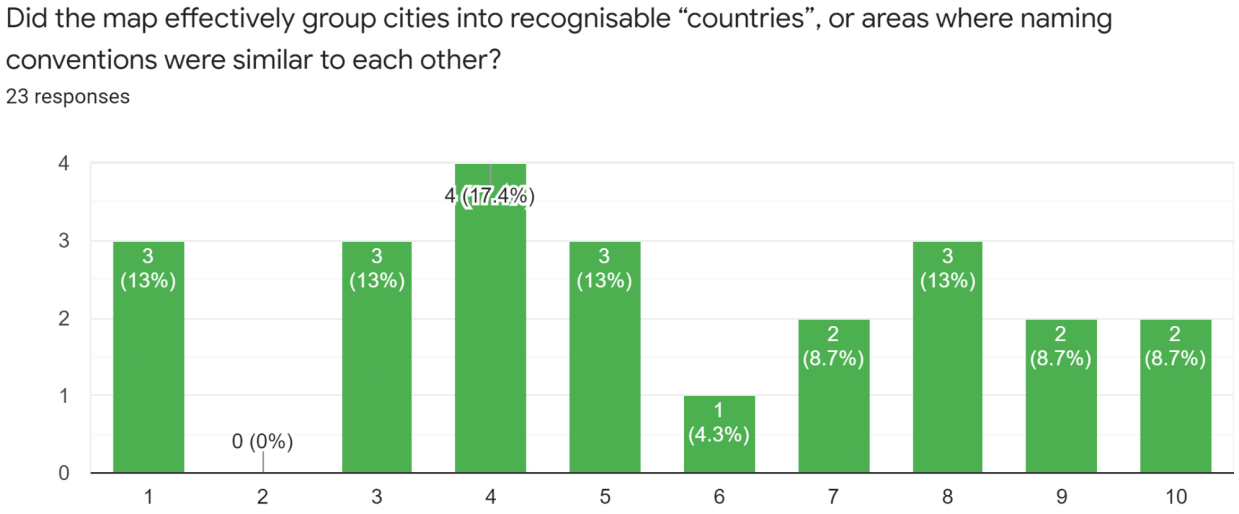
Question 4



Question 5



Question 6

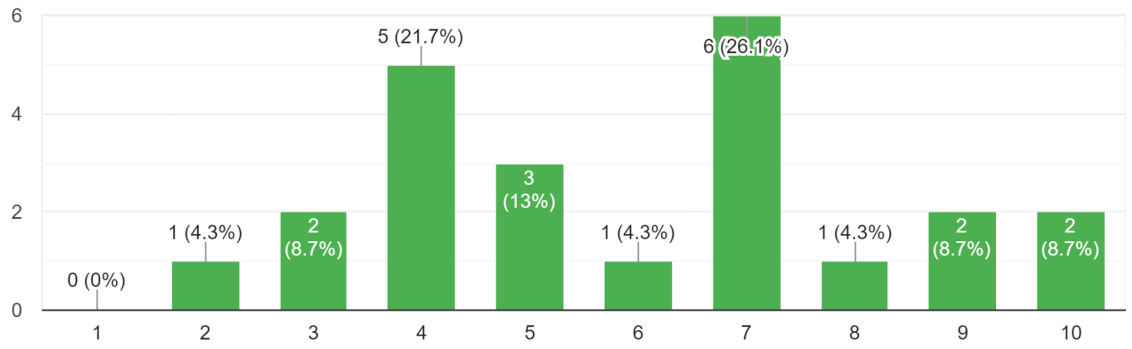


Map 2

Question 1

How believable do you find the layout of the map?

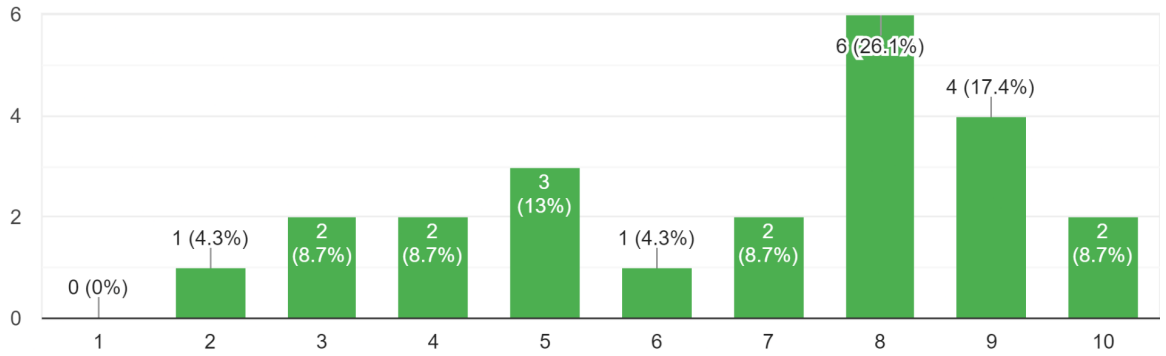
23 responses



Question 2

How well-spread were the city markers across the map?

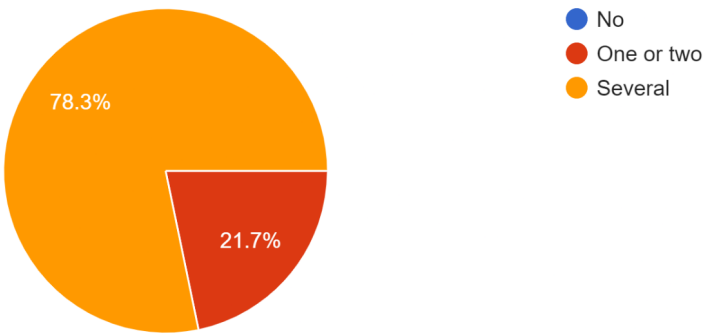
23 responses



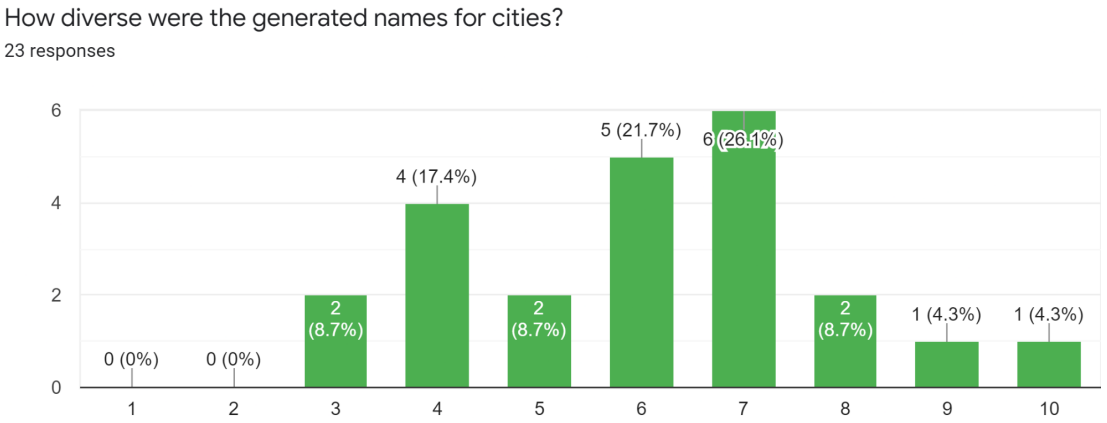
Question 3

Did you notice distinctive biomes in the map? (Areas of different colour of terrain, like snow, sand, grass etc)

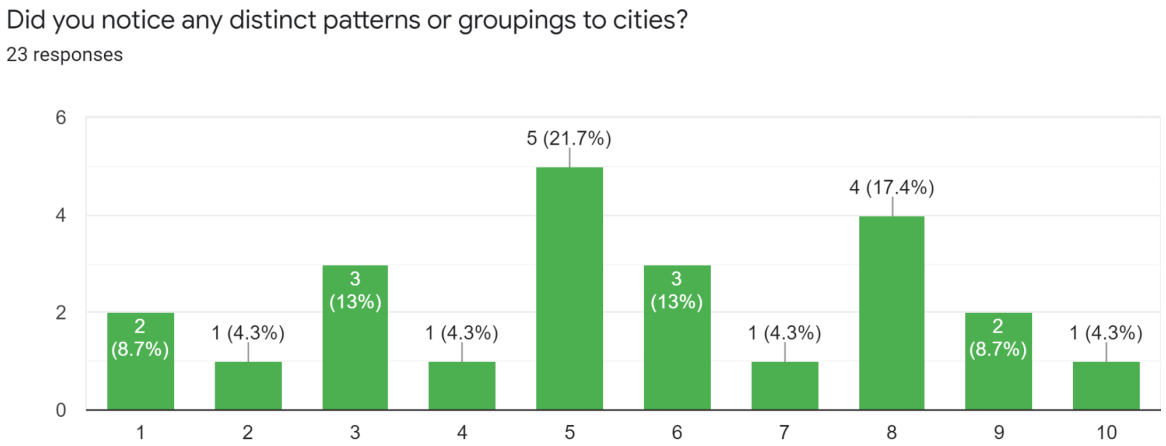
23 responses



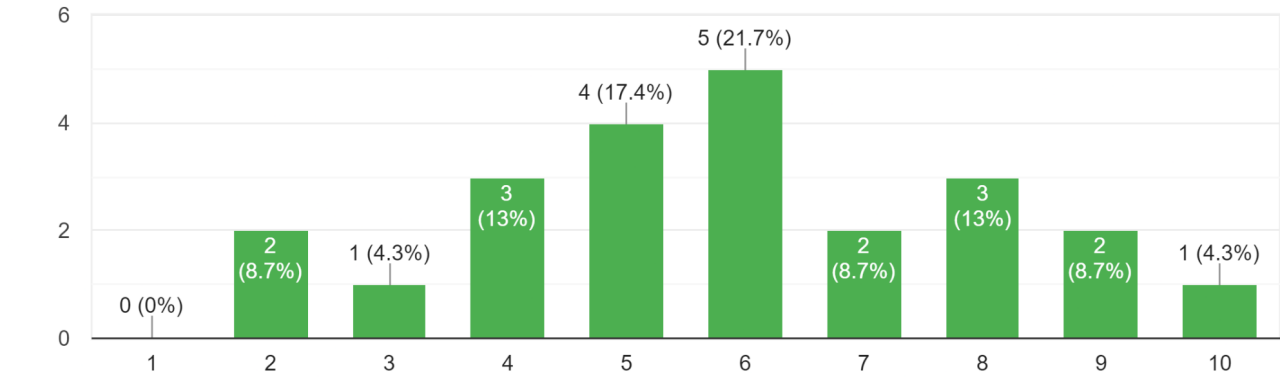
Question 4



Question 5

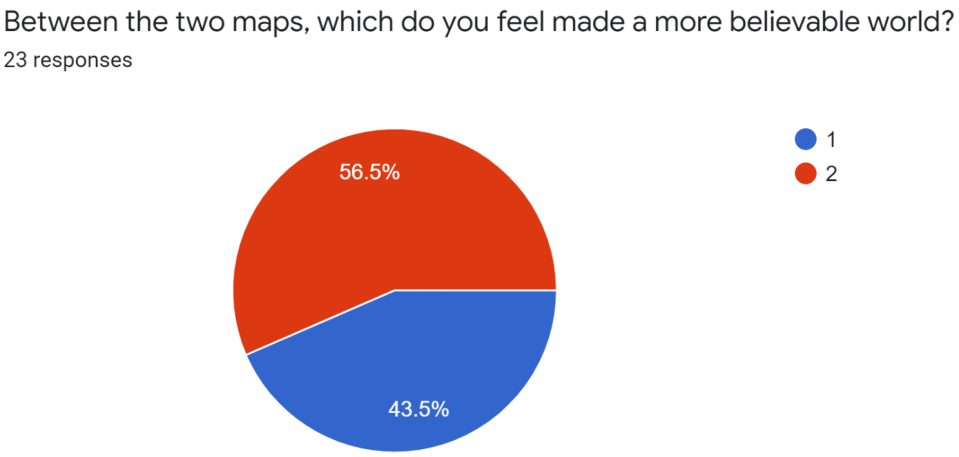


Question 6



Final Questions

Question 1



Question 2

Why do you think this? Please explain the specific differences that give you this opinion (terrain layout, city names, city placements etc?).

The cities in the second map were distributed slightly more evenly, and the names were more noticeably grouped according to how similar they sounded.

It was the metric I used to judge the other map. The names were more diverse and the cities were slightly better spread apart.

Appeared to be less cities on the map and there was greater distance between cities which seemed more realistic to a real world map. Some cities appeared in the middle of bodies of water.

Looked more akin to a fantasy world than the second one did, too sporadic the second time round.

several of the cities in this map were placed near bodies of water which makes for more believable city placement. there were also no cities build on top of snow-capped mountains, which is sensible

It looked better I think

There are some cities that are grouped together with a similar naming structure

Felt like it gave more cities across a range of biomes like you would expect in a real world setting. City placements were scattered out the map but near one side of the map there were some closer and some spread out, again more like a real world setting would be. Was more of a pattern of city names being similar when close together.

More landmass looks closer to an island instead of random rivers and lakes

Some very close clustering of cities with similar names on a desert biome with mostly water surrounding it, and on the opposite side of the water mostly grassland with cities there with similar names. Makes it look like a small country surrounded by a larger country.
Both maps were too lake/dessert heavy, but map 2 had more water/lakes in a way that seemed unrealistic.

(For overall quality question), in current state no as it changes between biomes so frequently it doesn't look realistic, especially the large amount of desert/lake biomes. But with some tweaks so there's less lakes/desserts & looking a little nicer it could work!

Less cities in the water.

Pretty much all of the above, but mostly the city placements!

1 had what seemed to be an excessive amount of water and at least one instance of the same city name being repeated with different spellings that wouldn't affect pronunciation (Savah vs Saavah vs Savaah) three times. Map B also had lots of similar names, but roughly split into three 'country' groups, whereas Map A's cities were more diffuse with no clear space between different naming conventions.

The terrain was more spread out with many city names being different from each other and not sounding the same.

The first was way more patchy, there was honestly a bit too much water, and too many towns in water, as well as towns spread too evenly. Also too many names in a row that were too similar.

The terrain was much more realistic in map 1, map 2 was a bit more chaotic in it's layout

Terrain layout for the second map was more natural and had a more realistic feel.

More believable names and cities placement was more realistic the first time.

Map 2 had more localised biomes, map 1 seemed more fractured.

The cities in map 2 were clustered, making it appear like they were part of different nations. Map 1 was fairly evenly spread out.

City names were more appropriate

Biomes were spread more realistically and city names weren't as repetitive

The names/groupings in map 1 were far more clustered and had 2 or 3 similar prefixes for all the cities.

Map layout was more fluent

Question 3

Across both maps, did you notice the naming themes? And did you find them interesting? Please explain your answer.

I did not notice them until prompted by the survey, but once I was looking for them, I could see some broad themes. I found the individual names interesting (they largely felt very plausible for a fantasy world), although looking at the map as a whole, too many of the names were far too similar to one another.

Lots of Zs, Ss, and Ts

The names seemed interesting but not sure of the theme.

Names were good, perhaps towns lying next to rivers/lakes could have more Gaelic pronounciations instead of the seemingly generic fantasy names. However, depending on the world it is set, this could be deliberate.

cities grouped closely together do look like they share similar names as if they share a similar language/ culture and naming scheme.

A lot of names start with A,S,T or Z

No I got tired of seeing the same letters

They began with A, B, S, T & Z

Yes. I found from some of the names it would reflect the biome it was in but then others just seemed completely random so I wasn't sure what tied that place to that naming convention.

Yes, no they are not interesting and sound generic fantasy names.

I thought the cities were named around the type of country they were with, but after this question it made me realise that they're actually named on the type of biome they're on? Definitely a cool idea but after spotting that it made it harder to tell what was a country boundary (if there is any at all) and what was just random biomes next to each other.

There did seem to be some similarity although it was inconsistent. The fact they seem to be made from procedural string chunks was nice, but the chunks could use more variance to enhance distinctions between areas. It was difficult to determine individual "countries".

Somewhat, but wasn't quite distinctive enough to tell for certain. I like the names regardless, pretty believable for the most part.

Lots of 'sharp' sounds like T, V and Z, as well as lots of S's. Also, all city names began with T, Z, A or S, and there's a distinct lack of 'softer' letters like P, C and M. It's an interesting stylistic choice on the part of the creator - if I was to speculate, I would suggest that it was designed to create somewhat harsh-sounding names, possibly suitable for cities named by a classically 'primitive' species. Perhaps lizardmen, since the S and Z are evocative of hissing, and guttural sounds like 'kh' and 'gh' or blunt ones like B and D (which one might expect from orcish city names) are not used. But it also places harsh constraints on the overall variety of names and leads to many of them sounding very similar, to a degree that I feel would cause confusion among hypothetical players ('was the capital city Tordarak, Torverak or Torveren? I know I've been to all of them recently...').

Yes, depending on the area of the city it would reflect the current terrain surroundings, for example cities around snow areas all started with the letter S.

There's a handful of prefixes and suffixes that are noticeable, seems to generally have to do with which biome they are located in roughly. Couldn't see any noticeable trend in them iddle part, besides light repetition at times.

I did notice them, they were interesting but could be a bit confusing as to how the conventions came to be.

I found that cities spawned on a specific biome would have a name more similar to others of that biome, and on the second map, cities close together often had very similar names - I had Tulbadadh, Tulvaadh and Torvaadh all clumped together

I did indeed notice themes along the names based on the city location, more than interesting some were very absurd to believe they were cities, but probably our planet is full of unrecognizible city names and I am just assuming after all.

The names all started with "A, B, S, T, or Z"

Most city names have 3 syllables

Sort of, I could see where prefixes were similar but they weren’t always grouped togetger

I didn't notice any themes, but names sounded realistic, especially if implemented for a fantasy game

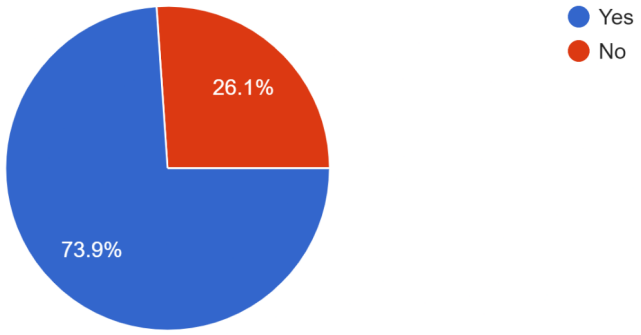
I noticed common letters for the beginnings of most of the cities in each map, though few similarities otherwise. Each city read like a real place name.

about 2/3 of the cities began with 'Tor' the ends of the cities was quite diverse though.

Question 4

Overall, do you feel this is an effective map generation tool (did you feel the maps generated could be used, or were they not good enough quality)?

23 responses



Quantitative Results by Tool Familiarity

			Respondents familiar with tools					
Response	Know overworld maps?	Believable layout?	Well-spread city markers?	Notice biomes?	Diverse names?	Patterns to cities?	Countries?	Thought it was effective?
1.1	10	3	8	Several	4	1	3	
1.2		3	9	Several	5	1	6	No
2.1	10	7	7	Several	8	5	8	
2.2		7	5	Several	6	5	5	Yes
3.1	10	4	5	One or Two	7	2	5	
3.2		4	7	One or Two	7	5	5	No
4.1	10	4	8	Several	7	7	9	
4.2		4	9	Several	7	5	5	No
5.1	10	4	4	Several	2	4	4	
5.2		4	5	Several	3	3	4	No
6.1	9	8	4	Several	2	2	4	
6.2		9	7	Several	4	4	4	Yes
7.1	10	4	8	One or Two	3	7	7	
7.2		7	8	Several	4	9	8	Yes
8.1	10	6	7	Several	7	1	1	
8.1		7	5	Several	8	2	2	No
9.1	3	5	8	Several	9	7	5	
9.2		4	10	One or Two	6	3	4	Yes
10.1	10	10	10	Several	10	3	3	
10.2		10	10	Several	10	3	3	Yes
11.1	8	6	5	Several	3	5	4	
11.2		2	2	Several	6	5	6	Yes
Average	9.090909091	5.545454545	6.863636364		5.818181818	4.045454545	4.77272727	

			Respondents unfamiliar with tools					
Response	Know overworld maps?	Believable layout?	Well-spread city markers?	Notice biomes?	Diverse names?	Patterns to cities?	Countries?	Thought it was effective?
12.1	10	8	7	One or Two	5	6	7	
12.2		7	8	One or two	6	6	5	Yes
13.1	8	5	5	Several	4	2	6	
13.2		5	3	Several	7	6	2	No
14.1	10	6	9	Several	8	10	9	
14.2		5	8	Several	7	9	9	Yes
15.1	10	4	4	One or two	5	6	4	
15.2		4	3	One or two	4	5	6	Yes
16.1	10	8	8	Several	7	8	1	
16.2		7	4	Several	7	1	6	Yes
17.1	7	8	6	Several	5	8	8	
17.2		5	8	Several	5	7	7	Yes
18.1	3	8	7	Several	8	8	10	
18.2		9	9	Several	9	6	6	Yes
19.1	10	3	4	Several	4	9	1	
19.2		3	8	Several	3	10	9	Yes
20.1	10	5	8	Several	4	8	5	
20.2		8	4	Several	4	8	8	Yes
21.1	10	10	10	One or two	10	8	10	
21.2		10	8	One or two	8	8	10	Yes
22.1	9	6	7	One or two	5	6	8	
22.2		6	6	Several	6	8	7	Yes
23.1	10	4	7	Several	4	8	3	
23.2		7	9	Several	7	8	8	Yes
Average	8.916666667	6.291666667	6.666666667		5.916666667	7.041666667	6.45833333	

References

Allen, I.E. and Seaman, C.A., 2007. Likert scales and data analyses. *Quality progress*, *40*(7), pp.64-65.

Berente, N. (2021). *Procedural Generation is an Art*. [online] Available at: https://designingwithmachines.com/2021/03/12/procedural-generation-is-an-art/ [Accessed 17 Nov. 2021].

Hendrikx, M., Meijer, S., Van Der Velden, J. and Iosup, A., 2013. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, *9*(1), pp.1-22.

Chen, B., Havelock, D., Plante, C., Sukkarieh, M., Semeráth, O. and Varró, D., 2020, October. Automated video game world map synthesis by model-based techniques. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (pp. 1-5).

Donjon.bin.sh. 2022. *donjon; Fantasy World Generator*. [online] Available at: <https://donjon.bin.sh/fantasy/world/> [Accessed 12 March 2022].

Game Rant. 2022. *The Elder Scrolls 6's Procedural Generation Has Huge implications For Its Map Size*. [online] Available at: https://gamerant.com/the-elder-scrolls-6-map-big-good-procedural-generation/ [Accessed 20 April 2022]

Ganiev), A., 2022. *Azgaar's Fantasy Map Generator*. [online] Azgaar.github.io. Available at: <https://azgaar.github.io/Fantasy-Map-Generator/> [Accessed 10 March 2022].

Kennard, S., 2019. *A beginner's look at using Perlin Noise for procedural map generation*. [online] Medium. Available at: https://samkennard.medium.com/a-beginners-look-at-using-perlin-noise-for-procedural-map-generation-4b8c7f6d2977 [Accessed 16 March 2022].

Patel, A. (2010) *Polygonal Map Generation for Games.* [online] Available at: https://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/#future [Accessed 20 Apr. 2022]

redblobgames.com. 2015. *Making maps with noise functions*. [online] Available at: <https://www.redblobgames.com/maps/terrain-from-noise/> [Accessed 21 April 2022].

Shaker, N., Togelius, J. and Nelson, M.J., 2016. *Procedural content generation in games*. Switzerland: Springer International Publishing.

Short, T. and Adams, T. eds., 2017. *Procedural generation in game design*. CRC Press.

Smith, G., 2015, June. An Analog History of Procedural Content Generation. In *FDG*.

Thirslund, A. 2017. *PERLIN NOISE in Unity - Procedural Generation Tutorial*. [online] Available at: <https://www.youtube.com/watch?v=bG0uEXV6aHQ&t=349s> [Accessed 12 March 2022].

Thirslund, A. 2017. *GENERATING TERRAIN in Unity - Procedural Generation Tutorial*. [online] Available at: <https://www.youtube.com/watch?v=vFvwyu_ZKfU> [Accessed 12 March 2022].

Thirslund, A. 2017. *PROCEDURAL TERRAIN in Unity! - Mesh Generation*. [online] Available at: <https://www.youtube.com/watch?v=64NblGkAabk> [Accessed 12 March 2022].

Togelius, J., Champandard, A.J., Lanzi, P.L., Mateas, M., Paiva, A., Preuss, M. and Stanley, K.O., 2013. Procedural content generation: Goals, challenges and actionable steps. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Togelius, J., Yannakakis, G.N., Stanley, K.O. and Browne, C., 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(3), pp.172-186.

Togelius, J., Yannakakis, G.N., Stanley, K.O. and Browne, C., 2010, April. Search-based procedural content generation. In *European Conference on the Applications of Evolutionary Computation* (pp. 141-150). Springer, Berlin, Heidelberg.

Unity Technologies, 2022. *Unity - Scripting API: Mathf.PerlinNoise*. [online] Docs.unity3d.com. Available at: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html [Accessed 21 April 2022].

Van Der Linden, R., Lopes, R. and Bidarra, R., 2013. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(1), pp.78-89.

Whittaker, R.H., 1962. Classification of natural communities. *The Botanical Review*, *28*(1), pp.1-239.

Yu, D., 2016. *Spelunky: Boss Fight Books# 11* (Vol. 11). Boss Fight Books.