**Introduction**

The task at hand was to create a procedurally generated 3D role-playing game. Essentially it would entail what a normal role-playing game would contain but with the world and many of the parameters of said world being procedurally generated. Procedural generation is a method of creating data algorithmically as opposed to manually. The reason for creating such a game would be to provide a unique player experience multiple times over, by providing a world with which each playthrough is different, with different biomes, foliage, characters, structures and more. It’s an experience that can revisited multiple times and still not become boring and obsolete, a clear example of this is Minecraft, a game released November 18, 2011, becoming the most sold game worldwide, with over 200,000,000 sales, and even though it’s a voxel based and doesn’t use a game engine, it uses the Lightweight java game library, it’s seen consistent success over the course of more than a decade.

Besides the fact that each generated world is unique, a lot of procedurally generated games are nigh-endless or at the very least so large that you could never hope to explore all of it. However, even if every playthrough is unique, most of the time, procedurally generated games lack the finesse and polish that a lot of premade worlds and games have. Since in non-procedurally generated games, there aren’t as many variables you have to control and maintain, the game world is always the same, but because of that it’s easy to improve on what you can see, its harder to modify an algorithm, since the outcome’s always going to be different, so you can try to constrain values and amplify others. But, if the smaller than of possibilities for a certain parameter, the less unique the experience is overall and therefore the more repetitive the game becomes, taking away from one of the largest benefits and strengths from a procedurally generated game. If there’s enough different about each world, the replayability of a procedurally generated game should nearly always triumph over a premade one, since usually a premade games content is often more constrained.

To add onto, even if a procedurally generated world is infinite and ever changing, you’re not going to go exploring it if there’s nothing out there to find. Even games like Minecraft have other features to support the procedurally generated world, since even in a such a game there’s still only so much you can do, to support this Minecraft have building, crafting, trading, mods and plugins that allow for so much more replayability, there’s so much more than just the procedurally generated aspect. Same with games like No Man’s Sky with its space style combat system, resource collecting and upgrade system. When paired with other systems like crafting systems, structure generation and combat, the amount of variables and parameters that can be procedurally generate increases enormously, way more than you ever could without, the things there can be in your world, the more variations of those objects that can exist increases.

On the other hand, creating procedurally generated games can be an endeavour in of themselves. I mentioned earlier that it’s harder to polish certain parts of a procedurally generated game, as although you’re polishing what ever you’re procedurally generating, to be more precise, you’re polishing the algorithm that generates said object, a lot of the time, procedurally generated assets are lacking in quality compared to their non-procedurally generated counterparts, due to this reason. On occasion, to get the desired result consistently, even for smaller, less noticeable features, it can take a considerable amount more time than would have originally been expected, as compared to a non-procedurally generated game, to get the desired result, since you’re either editing the algorithm directly or the parameters that its intakes. However, sometimes even though the outcome isn’t necessarily what you expected, it can still be desirable, the algorithm can produce an interesting result, since it’s hard to visual what something’s going to end up looking like just from examining the algorithm. But, because of this, it means it can often be much harder to predict how long certain parts of a project are going to take to develop, way more than if it was being manually made.

At times, in often extreme cases, some parts of the game can end up being unplayable, or in less extreme cases, produce an undesirable result, decreasing the enjoyment/quality of some parts of the game. It’s obvious, but the amount of things that could go wrong is the same as the amount of things that you need to go right, for instance, if you generate a part of the terrain and it looks aesthetically beautiful, but it’s been generated in a way that makes it impassable (assuming it was made to be passable), or even if it’s difficult, it is passable, but only for the player and not any AI. Of course, with due work and remaining true to the software development cycle, this eventuality should be mostly avoidable.

**Methodology**

The software development tool/game engine I’m using to develop my game is Unity. My reasoning for this is not just my own experience and familiarity with the platform, but also it’s user friendliness, it has a user friendly UI and is based upon it’s ECS (Entity Component System) and job system. The ECS is easy to comprehend and is as the name implies, a system made up of entities, which have components which tell those entities how to behave. The job system from the Unity3D documentation is described as a system that “lets you write simple and safe multithreaded code that interacts with the Unity Engine for enhanced game performance.” The reason I believe this to be pivotal, is that a large amount of my game is going to be generated on the fly, multithreading allows certain “jobs” to processed at the same time, meaning that I can split up parts jobs that need to be done over multiple threads, since forcing these onto the main thread wouldn’t be possible for pretty much any computer. Unity also has a variety of debugging tools, for optimising your game. It also has a large asset store, and although all the logic will be made by me, I can source the necessary models I need from there and even get some inspiration. There’s also a large amount of official documentation on each component and aspect of the game engine and even if that isn’t enough, due to the engine being free to use, there’s a large amount of community documentation, videos, and discussions on even very specific topics.

The way I set up my schedule was in a way that sure that for each part of the game, any prerequisites were already in place, e.g. I couldn’t start placing trees and foliage without there being any terrain. Since I’m endeavouring into a topic I have essentially no experience in, I’ll be doing my best with trying to adhere to the software development cycle. First comes planning, although I’m familiar with unity, I’m unfamiliar with the job system and multithreading in general, but not just that, procedural generation isn’t a field I’ve researched much, so I’ll be spending a decent amount of time researching what a certain aspect of the game would entail, this isn’t necessarily Unity specific however. The schedule that I’ve come up with defines broadly what I plan on doing, with minimum requirements as to not disrupt what needs to be done next. Next comes analysis, this’ll be where I’ll be deciding what from my research I actually use, since there’s limitations to what each computer can do, but also remove some of the unnecessary parts, to make sure I’m as time efficient as possible. When it comes to design, this’ll be Unity specific and what I’m able to, but also what parts I’ll need, essentially deciding what assets will be necessary and what scripts will need to be written, but also a decent part of this will be deciding what will go onto the main thread and what will be put onto Unity’s job system. After that comes implementation, majority of this time will be spent programming, it’s likely I’ll constantly be going back and forth from implementation and testing, since it’ll be hard to tell once I’m actually getting the desired result. After that of course comes testing and the main two outcomes I’ll be looking for are qualitative and quantitative results, essentially looking for whether there’s enough of something appearing or occurring and whether it’s occurring in the desired way. An example of this would be, if the right number of trees are appearing for a certain biome, but also that they’re being spawned in a way which doesn’t cause the system to lag and that they’re being spawned in a way that doesn’t disturb the players immersion. Maintenance for will mostly be that as other features are implemented, this system still works and neither system interferes with the other.

**Terrain Generation**

**Planning & Research**

The main way of generating terrain procedurally is using noise, noise can be though of a map of points, with each point being a number from 0 to 1. This can be used in several ways but for terrain generation it’s not so complicated, as for simple terrain generation you only really need a height map, which is essentially what this is. There are multiple different types of noise, the three I’ll be looking at are Simplex, Perlin and Voronoi noise.

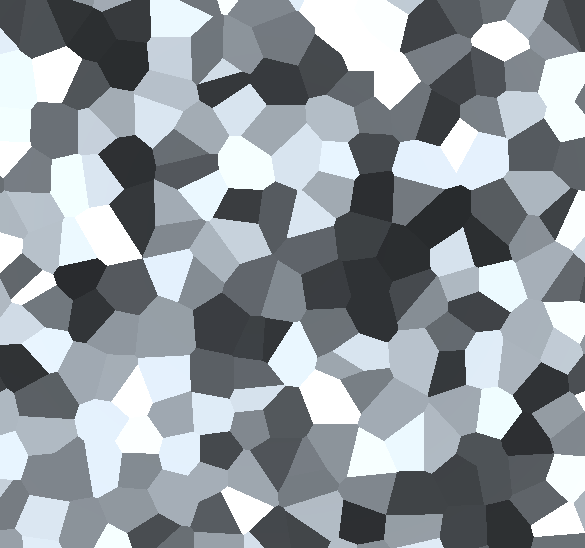
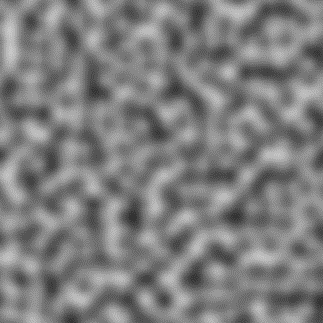
Perlin noise is described a complex algorithm for generating coherent noise. This type of algorithm is used quite a lot for terrain generation because the noise it generates is in the form of a gradient, so it can either, be used directly for creating smooth continuous, unending terrain or be used a base for the terrain. It is also worth mentioning that Unity has a built-in function for Perlin noise, so there would be no need to any implementation of a custom Perlin noise function.

Simplex noise used triangles instead of a square grid and is much harder to implement than Perlin noise, but can be saved for additional features, it also isn’t gradient based, meaning that each point is either 0 or 1, unlike Perlin noise which is a range from 0 to 1.

Voronoi noise places seeds and assigns each one a different biome, each point is assigned to the closest seed.

Simplex, Perlin, Voronoi

A picture containing fabric

Description automatically generated

**Analysis**

For the purposes of generating terrain, simplex is essentially just an inferior version of Perlin noise and although Voronoi noise would generate evenly sized biomes, getting a height map from it would be difficult, or at least more difficult than Perlin noise, mainly because Unity has a built in Perlin noise function. So at least for now, since it allows for upgradability later, but also for the purposes of being able to focus on other features, such as foliage placement, Perlin noise will be with what I go with.

**Design**

Since Unity has a built-in Perlin noise function that doesn’t need to be developed, however instead of simply assigning biomes based on height, I’m going to layer two layers of Perlin noise, one for the height of the terrain and one for temperature. Many of the implementations of Perlin noise I’ve seen usually overlap more than just two layers, but since this’ll be very easily upgradable in the future I’m simply going with two layers, one for height and one for temperature.