An Interactive Terrain Generation Algorithm in Unity

Brandon L Mendoza-Gaytan
Earlham College
CS488
blmendo22@earlham.edu

Abstract

This paper presents my capstone project, which outlines an interactive terrain algorithm implemented in Unity. I did this to create a visual learning resource. I will introduce the general field of procedural generation, provide a survey, and finally, lay out the implementation for my project.

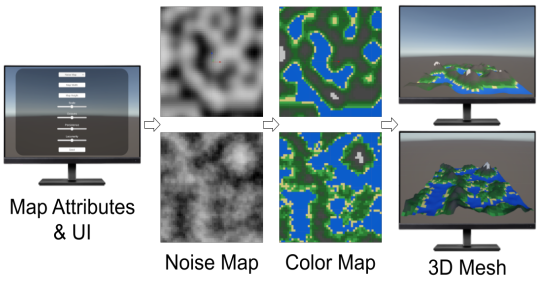


Figure 1: Graphical Abstract

1 Introduction

When you hear procedural content generation (commonly abbreviated as PCG), you probably think of video games. Indeed, it has its uses in other forms of media, such as movies, animation, and visual effects, but video games utilize PCG more than any other digital medium. PCG is nothing new. Games like Beneath Apple Manor (1987) and Rogue (1980) are among the first to utilize PCG to generate their content. Since then, PCG has grown both in complexity and reach, as well as in research and development. PCG is inherently a complex and computationally intensive algorithm that requires significant computational power, and research primarily focuses on making it more efficient or expanding its applications.

2 Review

2.1 What is Procedural Generation?

In video games and video game development, two primary methods exist for creating a video game's content: painstakingly handcrafting these elements or having the computer generate them through an algorithm. The latter is known as procedural generation. A PCG algorithm can be very complicated, but at its core, it is an algorithm that contains several rules that dictate what the result should look like. Minecraft uses it to generate its infinite number of worlds, as does No Man's Sky [5]. Spelunky utilizes PCG to generate levels while incorporating game mechanics, increasing replayability.

Figure 2: Minecraft and Spelunky

2.2 Handcrafting or Generating Content?

During the development of video games, a problem that all developers eventually encounter is creating content. Content can refer to various elements inside a game, such as levels. An algorithm or handcrafting can solve this problem [7]. Handcrafting content gives developers complete control over design, implementation, and deployment, but it comes at the cost of time and money. Depending on the content, it can take multiple people and groups to accomplish this.

Not to mention efficient collaboration and communication. Using PCG alleviates some of these concerns, at the cost of control over the content's more intricate designs. However, for some developers, this trade-off is worthwhile. [10]. I want to note that for some, PCG is not a replacement for handcrafting content but rather a tool to be used alongside human creativity [8].

2.3 Using Procedural Generation

As mentioned, PCG is used in video games and movies, and within these, in various areas, but it is used more in some than others. Liu Y. in [7] provides an excellent overview of PCG in action. Algorithms can be created to create different elements of a world. Procedurally generating vegetation is possible through an L system, a formal grammar system that lays out a potential shape for vegetation. An algorithm can produce the architecture of buildings and other structures to a believable degree. It's even possible to use PCG to program the behavior of entities within a game. Separation steering [2] is a unique algorithm created by Craig Reynolds to recreate animal behavior. He tailored it for boids, a birdoid object, and it mimics the behavior of a bird flock. These examples demonstrate how PCG is applied in practice and can be utilized across various media.

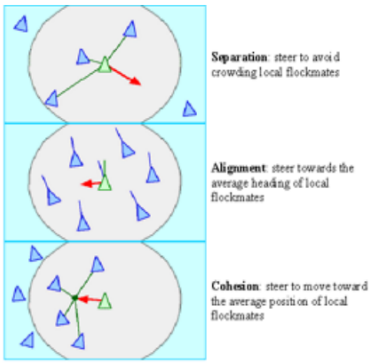


Figure 3: A visual representing separation steering

2.4 Techniques and Conventions

PCG has numerous applications in the real world, which means it encompasses a wide range of techniques and concepts that underpin it. Some are shared between applications, and some are not. In video game development, games that use PCG to generate terrain have similar processes. It all starts with a map of noise [11]. These maps form the basis of what the world will look like. A height map is created using this. Height maps determine the heights of various points in a world. The whiter a pixel is, the higher it will be when generated, and the opposite is true for darker pixels.

From here, the algorithm can go in many directions. Occupancy-regulated extension(ORE) is an algorithm meant to remedy the issue of most algorithms only working within their domain, that is, the specific environment to which they are tailored, taking into consideration game mechanics. ORE is more general and creates geometry without the limitations of the domain in mind. It is worth noting that this algorithm is intended for use in a 2D setting. Seeding [12] is a convention used universally within PCG. The seed is usually a string of numbers that plays a big part in the algorithm, as it is used as instructions to generate content. With this, it is possible to generate the same thing. Using seeds is typically recommended, as it helps with debugging and testing. Another fairly used convention is chunking. PCG is resource-intensive and becomes more intensive as you add graphics and other features. To minimize the load on hardware, you split the content, usually a level or terrain, into equally sized portions called chunks. Only a certain number of chunks are generated, loaded, and unloaded based on conditions specified by the algorithm. These are just some examples of concepts within PCG.

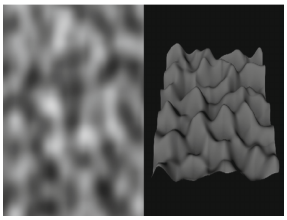


Figure 4: The left is a noise map, and the right is the result of using that noise as a height map

2.5 Problems with Procedural Generation

PCG is powerful, but it does not come without its problems. It isn't easy to implement PCG in a meaningful and practical way. The benefits that PCG offers assume a proper algorithm. Generated content typically looks generic [6], and integrating a PCG algorithm within its domain [8]

adds detail and character. Doing so adds complexity and more difficulty in creating a working algorithm, but this is practically a necessity to avoid a generic outcome. Otherwise, it may be worth creating content by hand. Another limiting factor of PCG is that there are very few general algorithms. What I mean by this is that few algorithms exist that can serve as a foundation and be applied to different projects. There are algorithms like ORE [10] which are made to be general and then built upon by others, but it is the odd one out. Any successful implementation of PCG is highly specialized and complex, making it challenging to translate to other domains.

2.6 The Future of Procedural Content Generation

Procedural generation's applications are numerous and can be flexible and tailored to any experience. Algorithms can be created that generate levels and terrains for video games, create structures and architecture, imitate animal behavior, and much more, bringing a world to life. As is the case with many technologies, PCG is headed in the direction of AI and machine learning integration. Trends for the future involve enhancing the content generated by using AI to make worlds, stories, art, and aesthetics more tailored to players' actions, making PCG accessible to users without requiring a deep understanding of algorithms, and to facilitate real-time generation that reflects a user's actions and decisions. The future looks promising. PCG is a powerful tool that can be used in conjunction with human creativity to enhance digital media and create a more engaging experience.

3 Procedural Level and Terrain Generation

I want to hone in on one PCG application: procedurally generated levels and terrain, which is commonly used in video game development. I want to distinguish between levels and terrain, as the former tends to be multiple small bits that make up a game, and the latter tends to be a much larger portion. Regardless, these both have the potential to be some of the most challenging and resource-intensive aspects of a game. I would be willing to go as far as to say that a game's quality is predominantly decided by the variety, depth, and overall quality of its levels. Minecraft is a game that has enormous replay value due to its use of PCG to generate unique terrain every time a player creates a world. Spelunky generates different levels and incorporates the game mechanics into the algorithm, leading to a longer enjoyment of the game. While PCG is not a requirement nor a guarantee that if you implement it, your game will be a hit, it can elevate it if used properly.

I will focus on terrain generation because that is the centerpiece of my senior capstone project. I decided to do this as a project because I've played games like Minecraft, which use terrain generation for their worlds, a feature the developers have dubbed "World Generation." I have always wondered how it works and decided to take this as an opportunity to learn and implement an algorithm. I will not be creating an algorithm on the same scale as Minecraft. That is an algorithm with years and years of development behind it. Instead, I created a simple one in Unity for the purposes of this project.

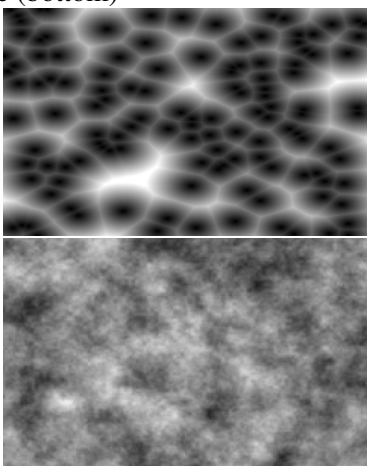
4 Design

I incorporated elements of user interactivity. I mentioned one of the issues that PCG has is the difficulty in understanding what is going on. This extends to procedural terrain generation. To help with this, I created an algorithm that users can interact with through a user-friendly interface. My inspiration came from Patel in [11]. Here, he adds interactivity and explains how noise, height maps, and other variables affect the terrain. I wish to do something similar.

First, I decided on terrain dimensions. It started off as a fixed size, but later added the ability to change the length and width. Next came choosing the kind of noise to use for the height map. I decided on Perlin noise, which is the most common type of noise map. However, many options exist, such as Worley, Simplex, Fractal, etc [1]. Initially, I wanted to implement the ability to change the noise used. Instead, I opted to have different modes that dictate what will be drawn to the screen. These modes are noise map, color map, and mesh terrain. I added adjustable noise parameters, such as scale, octaves, persistence, lacuarity, etc., for a more unique result. The resulting noise map is used as a height map.

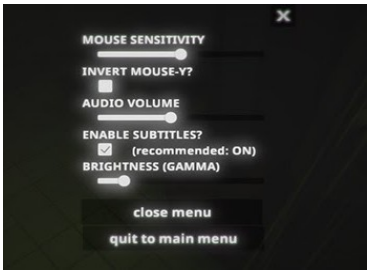
I wanted to create different biomes, but I de-

Figure 5: Worley/Voronoi Noise (top) and Fractal Noise (bottom)



cided to keep it simple and focus on color coding. For example, if the terrain is below a certain height, it will be water and blue. It will be a snowy peak and white if it is above a certain height. Everywhere in between would be green and other colors. The last major component was the mesh, which in this case refers to the land. This mesh accurately represents the noise map from which it was created and adapts its color to match the color coding previously explained. Lastly, there was the UI that could change the values of the algorithm itself. I wanted the terrain to change in real-time, without impacting performance, which was achieved. I added features and made adjustments throughout my work on this project, as well as testing and optimization.

Figure 6: Sliders in Unity, which will be used in my project



5 Contributions

When I researched PCG and the specifics of terrain generation, I found a substantial number of references to utilize. These references were papers from a journal, mainly because that was all that showed up. You had to specify if you wanted a PCG algorithm already implemented. GitHub has PCG and terrain generation projects, like [3]. You can find projects where people have experimented with and explored PCG, and some have created terrain generators that generate terrain based on fields specified by the user. However, I found these to be overwhelming. One resource that stood out to me was [11] because of its interactive nature. It started as text and images, but then came the first slider. The slider gave me control over the values of the code snippets. These snippets then controlled a 3D model that resembled the early terrain stages generated by a height map. I couldn't find anything like this on GitHub, so I took this idea and recreated it using Unity as a learning resource to introduce the concepts of terrain generation.

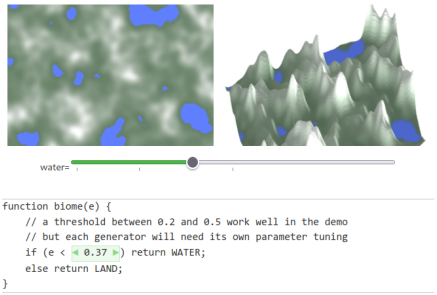


Figure 7: Here is a slider that changes the value that determines where the water level starts in [11]

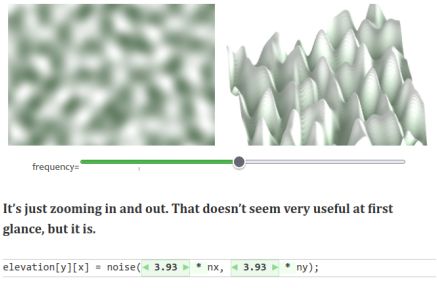


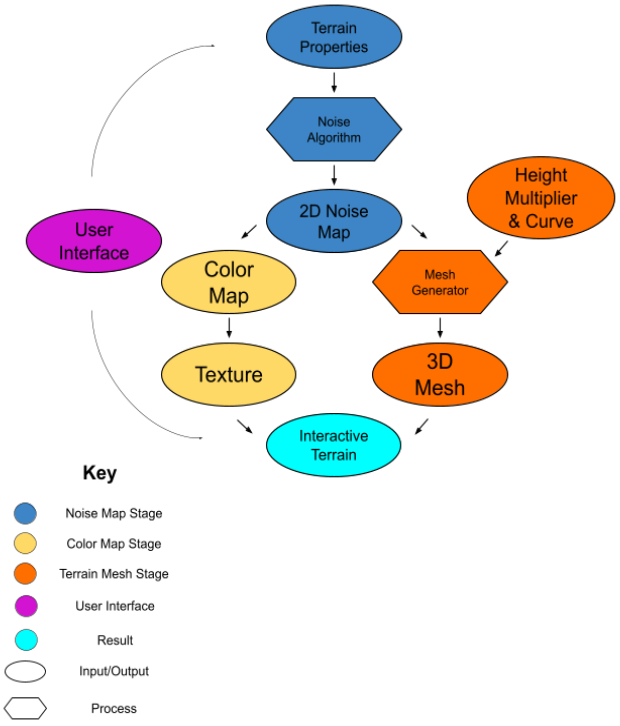
Figure 8: A slider that changes the frequency of hills in [11]

6 Implementation

6.1 Unity Version

When selecting a version of Unity, several factors needed to be considered. Selecting between Unity 6 or Unity 5, choosing between the newer

Figure 9: Data Architecture



universal render pipeline or the older built-in render pipeline, and selecting a long-term support build or the latest and greatest. What I have found is that, regardless of any combination of these, the programming remains largely the same. Where the differences start to show are with many of the graphical tools and features. I settled on a newer setup. I went with Unity 6 version 6000.2.2f2, the universal render pipeline, and a non-LTS build. The reason all this mattered is that some of my sources are on the older side. Their implementations of a noise map, mesh, and other things have outdated aspects. I could have gone with an older version, but I wanted this to be as up-to-date as possible and decided to tackle any issues that might arise from my using older methods on newer software.

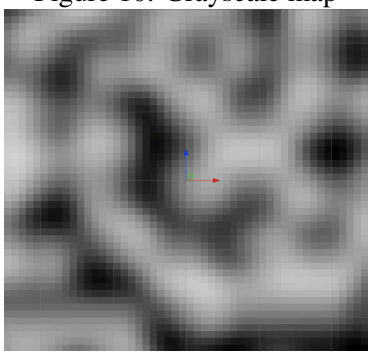
6.2 Project Structure

This project is split into different scripts. Each script is in charge of a single task: creating a noise map, a colored version of that map, a fall off map, generating a mesh using the noise map, creating a texture, assigning the texture to a renderer so that it is visible on screen, and the last script brings it all together. There is, of course, the user interface that the user will use to interact with the algorithm and view changes to the terrain in real-time.

6.3 Creating A Noise Map and Color Map

In Unity, creating a noise map is a relatively straightforward endeavor. In essence, it is a grid of float values. To create this, I needed to create a 2D array, loop through it, and for every element, use the built-in Perlin noise function to generate a float value between 0 and 1. The noise map will only exist as a structure in code, not visually in the gamespace. From here, I expanded and introduced more into the algorithm. Operating on the float value generated by the Perlin noise function brought in variation and new details. These values are the scale, amplitude, frequency, persistence, and lacunarity. Last but not least are octaves. Octaves are the different layers of noise. They, like all the other values, introduced new detail and more variety in the noise map. When all is done, you are left with a fairly detailed noise map, which is the foundation and will be used by almost every other script.

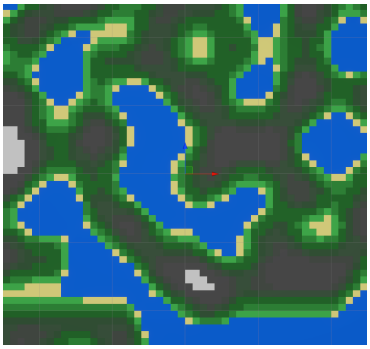
Figure 10: Grayscale map



A color map introduces color to the map generated by the noise map function. As stated, the map is an array of 2D float values and is not anything special yet. Two color maps are generated here—one in grayscale and one in color. The grayscale map adheres to the traditional scheme of noise, where the underlying noise map is represented using shades from black to white. The map in color is tailored to represent colors in a landscape: blue represents water, yellow represents sand, green represents grass, gray represents rocks and mountains, and white represents snow. Implementing a color map was straightforward. I created an array of colors that was the same size as the noise map. I evaluated the float

value at each index and assigned a color. If the float value were close to one, it would be white; if it were close to zero, it would be blue; and somewhere in between would be yellow, green, and gray. I looped through the noise map and did this for every value. The grayscale and color maps are only arrays at this point. To make them visible on screen, they had to be made into a texture, which is a simple matter of creating a new texture object, setting some options, and applying the color array to that texture. This results in two color maps that represent the underlying noise map. The grayscale map stops here, but the colored map goes on to be used for the 3D mesh.

Figure 11: A colored map. Matches grayscale map



6.4 Generating A Custom Mesh

Creating a custom mesh [4] was perhaps the hardest part of this project. There was a lot to learn here. Meshes in Unity and in general have a lot to them. Luckily, for my purposes, I didn't have to delve too deeply to find what I needed. Meshes in Unity require at least four things: vertices, triangles, a mesh filter, and a mesh renderer. There are plenty of additional components that can be added, but they are not required. We will need one called UVs, however. Vertices act as the frame or skeleton of the mesh. The remaining components build off the vertices. The triangles can be thought of as the skin. These triangles are constructed using the vertices through a traversal of them in a triangular pattern. UVs are how you can add a texture by telling Unity what parts of that texture go where on the mesh. The mesh filter holds these pieces of data, and the mesh renderer draws the mesh on the screen, making it visible.

Adding a mesh filter and a renderer is done simply by adding them as components in the editor. Vertices, triangles, and UVs are added through programming. These data values are all stored in arrays. Similar to the maps, I created an array that is the same size as the noise map and populated it with vertices. These vertices were created using X and Z coordinates (length and width) in the gamespace, made for easy debugging and testing, as they could be toggled to be visible. The Y coordinate (height) of the vertices was important here because every single vertex needed to correlate with a noise map value for an accurate representation of the original noise map. I set the height by passing the noise map as an argument to this mesh function, allowing the values to be accessed. I then set the height of the first vertex to the first float value, the second vertex to the second float, and so on. The final step was to apply any height modifiers, as the height is a small float value by default. With this, the vertices array was finished.

The creation of triangles involved groups of three vertices(because a triangle has three vertices). Traversal of the vertices was done in a clockwise order as specified in [4]. The order you traverse the vertices in determines the orientation of it. Clockwise traversal results in an upright mesh, while a counter-clockwise traversal results in an upside-down mesh. The triangles field is an array consisting of integers that correspond to a vertex's index in the vertices array. Where you start the traversal technically does not matter. You could start at the very center of the vertices. But in practice, it makes sense to start at the beginning, which is what I did. Figure 11 illustrates the process of creating mesh triangles.

With the triangles field populated, the final step was to set the texture of the mesh. Since the color array has already been applied to a texture, the texture can be applied to the mesh renderer, which will display it on the screen.

6.5 Creating A Falloff Map

One thing about the mesh is that once it reached the specified size, it ended. It didn't matter if it was in the middle of generating a mountain or a small island. This does not look natural, so I wanted to ensure that any piece of terrain could finish generating before getting cut off. This is an established concept called a falloff map,

Figure 12: Mesh triangles

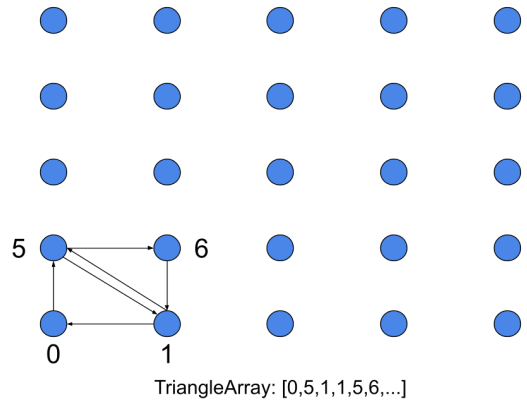
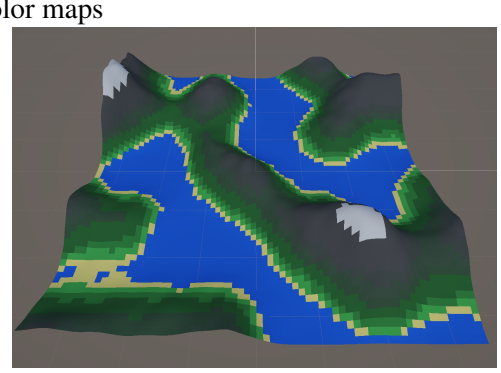


Figure 13: 3D Mesh. Matches grayscale and color maps

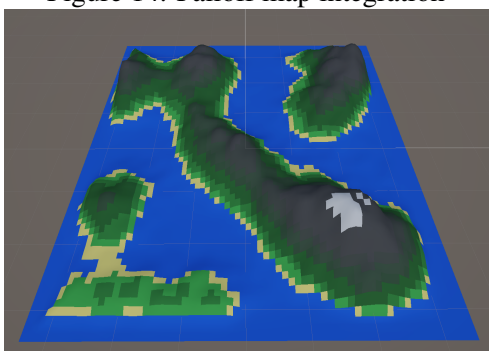


where the ends of the map are smoothed off before they become water. This involved creating a custom noise map. This map is an array of float values, but follows a different pattern than the Perlin noise map. In this custom map, the closer an index is to the center, the closer it is to 0, and the closer it is to an edge, the closer it is to 1. Then I subtracted these values from the noise map's raw float values. The modified noise map is one where the values approach or equal zero as its index is closer to the edge. With this, the terrain is being smoothed out at the edges so that none of it is cut off.

6.6 User Interface

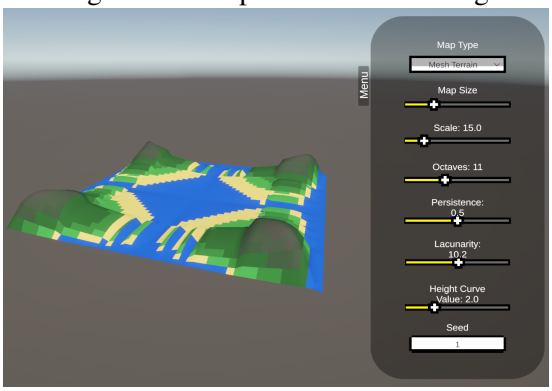
Creating A user interface for interacting with the terrain was mostly a straightforward process. Thanks to Unity's built-in tools and functions, implementing simple elements like sliders, drop-downs, fields, and buttons was no issue. I laid out UI elements in a canvas along with some labels. For each element, I added a listener that listens for when its UI element has

Figure 14: Falloff map integration



been modified. When it detects a change, it calls the script, which in turn calls all other scripts and functions to regenerate the noise map, color map, and terrain mesh. I did restrict the range of valid values to be used in the algorithm. This is because I could very easily crash the program if I set a value a little too high, such as 100,000 octaves. However, I left enough range for artifacting to occur so that the impacts of high values could be seen.

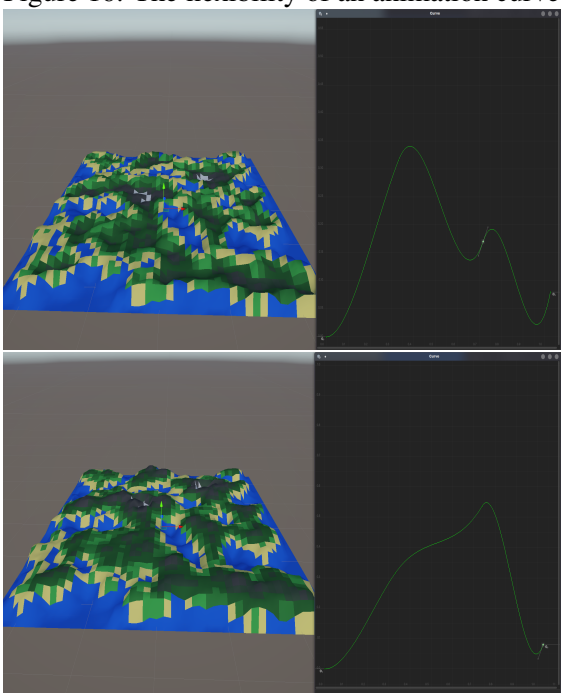
Figure 15: Simple UI and artifacting



There was one element that was tricky to implement. I added an animation curve in the editor, which is essentially an X-Y graph. This curve is used to control the height profile of the terrain by evaluating it against the height of the vertices in the mesh. For example, a curve with y=1 would mean that the height would be the raw height value of the vertices multiplied by the height multiplier. If the animation curve were shaped like a parabola, then the terrain would have exaggerated heights, but barely affected lows. The problem is that Unity does not provide a way to add this as a UI element, like a slider or a button. I had to create this in my own way. I did this by mapping a slider to modify an

exponent value where the base is the raw height value of a vertex. In this way, I was able to recreate the parabolic shape from the animation curve and achieve exaggerated heights, but fairly unaffected lows. However, this is not perfect as my implementation is restricted to exponential curves, whereas the animation curve can be manipulated into a wider array of forms, as shown in Figure 17.

Figure 16: The flexibility of an animation curve



7 Final Result

The final script is MapGenerator.cs. Here is where everything is brought together. This is attached to a game object of a similar name, map generator. This object holds all scripts and references to other objects, such as the camera and the UI canvas. It is essentially a sort of hub. All values for the noise map are taken from UI elements and passed to the map generator, which then passes them to the MapGenerator.cs script. The noise script is called and passed all the relevant values, returning a noise map. The noise map's dimensions are used to create a falloff map and a color map. I have an enum with the options 'Noise Map', 'Color Map', and 'Mesh Terrain'. This enum uses the drop-down UI element to determine what should be displayed on the screen. In any case, a texture will be created and then drawn to the screen. The 'Mesh Terrain' option creates a mesh and applies the texture to it before it is drawn on the screen. This process repeats whenever any noise map value is changed through the UI, which is what makes the terrain change in real-time. The in-game UI has most of the functionality of the editor interface. The advantage of the in-game UI is that it enables players to interact with the terrain without requiring the Unity editor or the source code to be downloaded. I also uploaded my project to Unity Play[9], where it can be played directly in a browser, eliminating the need for a download. In this way, I make it accessible. Figures 18 through 20 showcase my project, featuring some unique options set using the UI.

Figure 17: A unique grayscale map using the UI

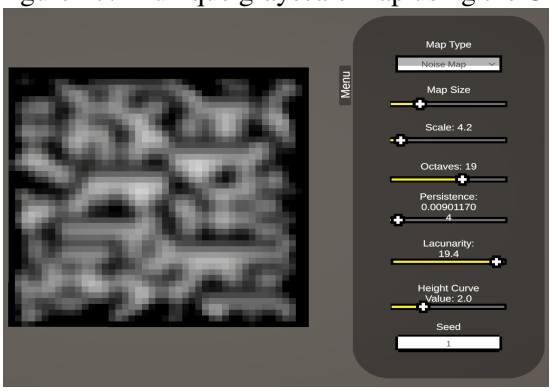
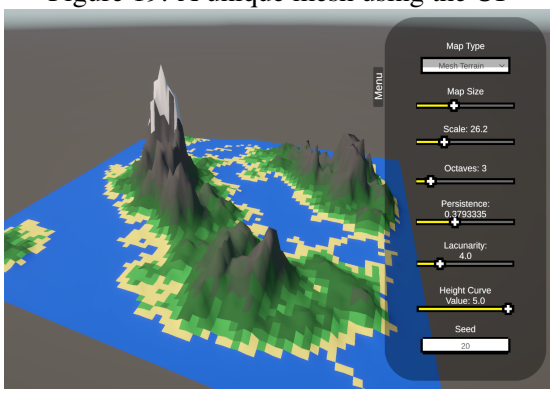


Figure 18: A unique colored map using the UI

8 Future Work

Overall, I am content with how this project turned out. The result is close to what I envisioned when I started planning and implementation. Some things could use work, however. The animation curve lets the terrain get more interesting because the odd curves can create unique height profiles. As mentioned earlier, my im-

Figure 19: A unique mesh using the UI



plementation is limited to exponential curves, which is only a small part comparatively. Making my implementation more robust is work that could be done. Other parts of a world could also be added. This project can also be expanded as a whole. I've created terrain here, but vegetation, animals, architecture, and a variety of other elements that a world might have can be added, along with a UI to modify the world on the fly. There are various directions to pursue for future work.

References

[1] A. LAGAE, S. LEFEBVRE, R. C. T. D. G. D. D. E. J. L. K. P. M. Z. A survey of procedural noise functions. *The Eurographics Association and Blackwell Publishing Ltd.* (2010).

[2] BOND, L. Procedural generation: An algorithmic analysis of video game design and level creationdesign and level creation. *Honors Theses.249* (2017).

[3] CLYENS, A. Procedural terrain generator. `https://github.com/aidan-clyens/TerrainGenerator`.

[4] DOCUMENTATION, U. Create a quad mesh via script. `https://docs.unity3d.com/6000.2/Documentation/Manual/Example-CreatingaBillboardPlane.html`.

[5] ENGELSTEIN, G. Procedural generation. *Gametek* (2023).

[6] J. TOGELIUS, A.J. CHAMPANDARD, P. L. M. M. A. P. M. P. K. S. Procedural content generation: goals, challenges and actionable steps. *ResearchGate* (2013).

[7] LIU, Y. An overview of procedurally generating virtual environments. *ResearchGate* (2024).

[8] MAWHORTER, P., . M. M. Procedural level generation using occupancy-regulated extension. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games* (2010).

[9] MENDOZA, B. Interactive terrain algorithm. `https://play.unity.com/en/games/afca6d41-5cec-4368-8f00-1558c5fa4b43/capstoneprojectv1`.

[10] NICHOLSON, N. Procedural content generation using occupancy regulated extension. *CS/DS student portfolios* (2018).

[11] PATEL, A. J. Making maps with noise. *Red Blob Games* (2015).

[12] SAMPAIO, P., B. A. F. B. . L. M. A fast approach for automatic generation of populated maps with seed and difficulty control. *ResearchGate* (2017).