

Furman University

- Department of Computer Science -

Ontogenetic and Teleological Algorithms for Procedural Terrain Generation

Examining two fundamental elements of procedural terrain generation in video games using Unity3D

Grayson Solis

December 2019

Sections

[1: Introduction 4](#_Toc27052885)

[2: Literature Review 5](#_Toc27052886)

[2.1: Limits of Traditional Video Game Development 5](#_Toc27052887)

[2.2: Procedural Content Generation 6](#_Toc27052888)

[2.2.1: Heightmaps 6](#_Toc27052889)

[2.2.2: Meshes in Unity 7](#_Toc27052890)

[2.3: Ontogenetic Algorithms 8](#_Toc27052891)

[2.3.1: Noise 8](#_Toc27052892)

[2.4: Teleological Algorithms 9](#_Toc27052893)

[3: Algorithms 10](#_Toc27052894)

[3.1: Examining Ontogenetic and Teleological Algorithms in Particular 10](#_Toc27052895)

[3.2: Ontogenetic Algorithm using Smooth Noise 11](#_Toc27052896)

[3.2.1: Algorithm 12](#_Toc27052897)

[3.3: Teleological Algorithm using Droplet Erosion 15](#_Toc27052898)

[3.3.1: Understanding Properties of the Droplet 15](#_Toc27052899)

[3.3.2: Algorithm 16](#_Toc27052900)

[4: Algorithm Combination Comparison 18](#_Toc27052901)

[4.1: Random Noise 18](#_Toc27052902)

[4.2: Smooth Noise 19](#_Toc27052903)

[4.3: Layered Smooth Noise 21](#_Toc27052904)

[4.4: Smooth Noise with Droplet Erosion 22](#_Toc27052905)

[4.5: Layered Smooth Noise with Droplet Erosion 24](#_Toc27052906)

[5: Conclusion and Future Work 26](#_Toc27052907)

[5.1: Choosing the Appropriate Procedural Terrain Generation Algorithm 26](#_Toc27052908)

[5.2: Future Work 27](#_Toc27052909)

[6: Appendix 29](#_Toc27052910)

[6.1: Heightmap Generator 29](#_Toc27052911)

[6.2: Noise Generator 29](#_Toc27052912)

[6.3: Droplet Erosion 29](#_Toc27052913)

[6.3.1: Droplet Erosion Heightmap Generator 31](#_Toc27052914)

[6.4: Layer Generator 32](#_Toc27052915)

[6.5: Noise Generators 32](#_Toc27052916)

[6.5.1: Noise Heightmap Generator 32](#_Toc27052917)

[6.5.2: Seeded Noise 32](#_Toc27052918)

[6.5.3: Smooth Noise 33](#_Toc27052919)

[6.5.4: Water Surface 33](#_Toc27052920)

[6.6: Height Color Map 34](#_Toc27052921)

[6.7: Terrain Mesh Generator 34](#_Toc27052922)

[7: References 36](#_Toc27052923)

# 1: Introduction

This research paper will examine two important types of algorithms that are frequently used in the process of procedurally generating landmasses for video games. The algorithms in question will be an ontogenetic algorithm that utilizes a simplified version of Perlin noise to pseudo-randomly generate terrain, and the second will be a teleological algorithm that attempts to simulate rainfall and modify terrain via erosion. The implementation section of this project uses Unity3D, a game development software, to apply combinations of these two algorithms and display them side by side within the program. Several tests will be performed, and will be documented in **Section 4** of this paper.

These two algorithms will be used in a variety of ways to generate land that is compared in terms of several properties. These properties include realism, processing power used during generation, ease of implementation, and the applicability that the generated land might have in a video game setting. The goal for this research project is to assist anyone interested in learning more about the benefit of procedural generation in video games, whether the readers be aspiring game developers or simply hobbyists. This analysis is important for anyone that plans to use procedural landmass generation, as the two algorithms chosen for the research are applicable in a myriad of ways, as will become evident in later sections of the paper.

# 2: Literature Review

To fully understand the algorithms that will be examined in this research paper, one must first become familiar with the nature of the gaming industry throughout the years, as well as the overall concept of procedural content generation. It is also important to review the underlying thought process behind the two algorithms to establish a baseline before going more in depth in later sections of the paper.

## 2.1: Limits of Traditional Video Game Development

The limitations of the worlds in video games have more or less been directly influenced by the technology available to game developers at the time. Throughout most of the history of game development, game developers would craft worlds completely by hand, placing assets in such a way that led to a fun and engaging environment for the player to participate in. This made sense, as the relative size of game worlds started off quite small: many game cartridges for the Nintendo Entertainment System (NES), for example, had only around 128 kB of ROM [1]. Even throughout most of the early 2000’s, game worlds tended to be smaller in size due to the increase in memory and resources games in three dimensions demanded.

However, technology constantly improves and is continuing to improve today. Thus, as processing power and the ability to store more content increases, so does the demand by players for larger, more detailed worlds to explore. Modern computers even have the potential to run games with virtually limitless worlds, and modern consumers seem to be increasingly interested in these types of games. *Minecraft,* for example, is a game with a virtually limitless world, and is one of the most successful exploration games, reaching unprecedented popularity since its initial release in 2009 [2].

## 2.2: Procedural Content Generation

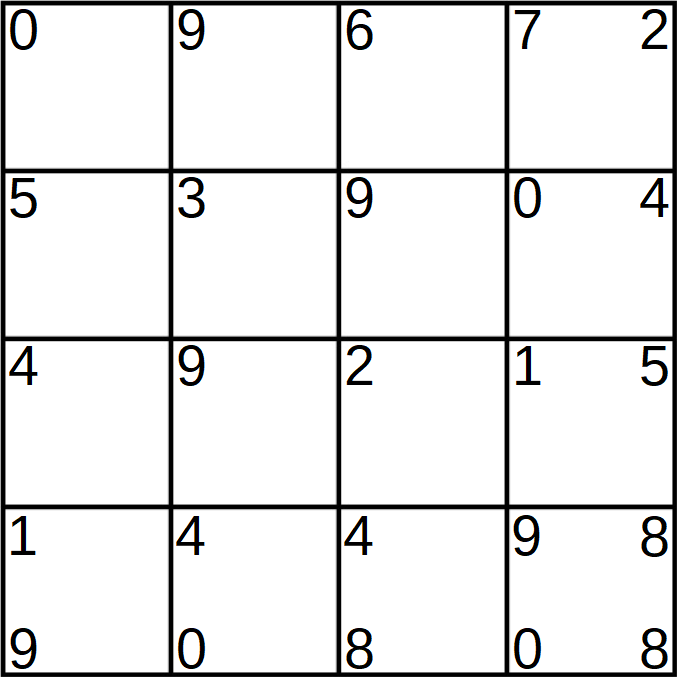
The question then arises: If constructing playable game worlds requires a team of programmers and artists, which in turn require to be paid for their work, how would it be possible to create games with infinitely vast worlds with a finite amount of resources? The answer lies within procedural content generation (otherwise abbreviated as PCG). This is a concept that assists game developers in the creation of vast amounts of game content, and allows for the creation of video games that would simply not be possible with a team of developers manually constructing the world themselves. Using a variety of different algorithms, oftentimes in tandem with one another, PCG uses pseudo-randomness to generate content with the goal of being as fun and interesting as content developed manually by a game developer.

There are many different kinds of content that can be generated using PCG. The level of difficulty, for example, can be adjusted by the game itself based on how often a player has died [3]. It can construct unique designs for creatures that the player can interact with, as can be seen in 2016’s *No Man’s Sky*, or even adapt the music based on what is currently occurring in the game’s narrative [4]. However, it can also be used to automatically generate landmasses, such as what occurs in games such as *Minecraft* and *Terraria.*

This research project will specifically be looking at procedural content generation in terms of how it can be used to generate landmasses, and breaking down two of the many algorithms that can be used for doing so.

### 2.2.1: Heightmaps

Heightmaps can be thought of as a grid where each vertex on the grid holds a value. A visualization of this is shown in *Example 1*. Heightmaps are a fundamental aspect of procedural terrain generation algorithms, as they are simplified representations of terrain that can be transformed into three-dimensional visuals by other components. In Unity, the component that can perform this transformation is called a mesh.



Example 1: Heightmap with a random value between 0 and 9 at each vertex

### 2.2.2: Meshes in Unity

A mesh is a three-dimensional object that is comprised of triangles, as can be seen in *Example 2*. Three-dimensional objects in Unity have meshes that can be manipulated to change the shape of the object. Each triangle on the mesh has three vertices which are stored in an array in the Mesh class.

These vertices are important, as the heightmap can be applied to an object’s mesh in order to transform the object into something that has the appearance of natural terrain. To achieve this, the number of vertices that correspond to a “square” on the mesh must be equal to the number of values that are contained within the heightmap (to better visualize this division into squares see *Example 3*). Then, each value in the heightmap can be inputted as a height value at every square’s vertices on the mesh.

A close up of a screen

Description automatically generatedA bathroom with a tiled floor

Description automatically generated

Example 3: How to break up a Unity mesh into a grid for use with a corresponding heightmap

Example 2: Mesh of a plane object in Unity; viewed from above

## 2.3: Ontogenetic Algorithms

When examining terrain in the real world, a sense of consistency is present. However, this is not to say that every physical object that exists in the real world is identical. Mountains, for example, are of course quite different from one another, and the same mountain of the exact same size is not simply copy and pasted in different locations in the world. Rather, terrain and landforms in the real world follow pseudo-randomness, where there exist similar features among all rivers, mountains, or plateaus, for example, but each of these landmasses is randomly shaped by tectonic plate shifting, wind, water, and other factors. In video games, ontogenetic algorithms procedurally produce terrain that mimics the real world by utilizing algorithms that produce noise [5].

### 2.3.1: Noise

To fully understand ontogenetic algorithms, it is imperative to first understand what noise is. Noise is a function that accepts an input value and generates a pseudo-random real number as an output value. The number generated by a noise function is in fact not truly random, because if the same input values are given to the function again, the same “random” output values will be generated. [6]

## 2.4: Teleological Algorithms

Ontogenetic algorithms may be thought of as a “something from nothing” approach, where the land generation is produced pseudo-randomly from a seed. Teleological algorithms, on the other hand, are best thought of as algorithms that modify existing land using simulated processes of the ones found in the real world [7]. These processes may include virtual rocks shaping landmasses, or simulated wind erosion. However, in this project’s case, a hydraulic erosion algorithm will be examined.

Hydraulic erosion is modeled by running simulated water across a terrain. This water is capable of two important actions that influence the terrain: soil acquisition and soil deposition. Soil acquisition will increase the amount of soil that the water is carrying with it and will decrease the height of the terrain under the point from which the soil was acquired. On the other hand, soil deposition will decrease the amount of soil that the water has in it and will increase the height of the terrain under the point at which the soil was deposited. Under what conditions the water will acquire and deposit soil, and the amount of soil that the water will influence at any given time will be covered in **Section 3.3.** Hydraulic erosion has one of the most important influences on the appearance of landmasses in the natural world [8]. Being able to effectively model this process will be a key aspect for this research project.

# 3: Algorithms

3.1: Examining Ontogenetic and Teleological Algorithms in Particular

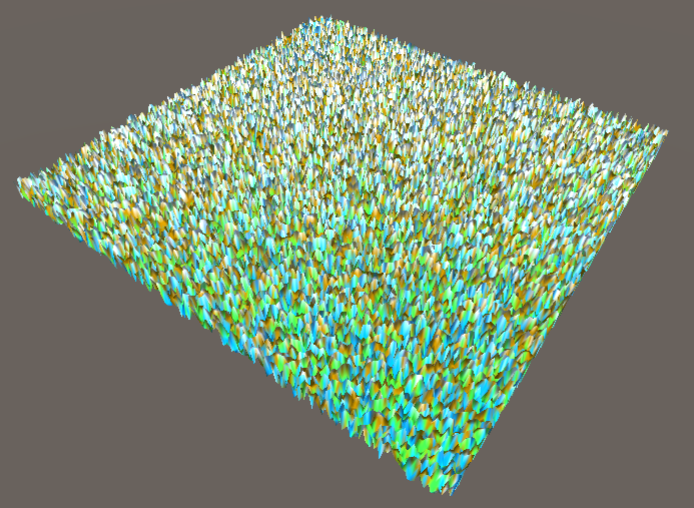
It should be noted that procedural terrain generation in games is not limited to the two algorithms that will be covered in this research paper. In addition to noise functions and hydraulic simulation, other novel approaches have been used for procedural terrain generation such as the use of software agents and evolutionary algorithms [5].

There are two fundamental reasons why this research paper examines ontogenetic and teleological algorithms in particular. The first is that, conceptually, these two algorithms are inverses of one another. Ontogenetic algorithms for terrain generation, as stated previously, aim to model natural environments using pseudo-randomness that constructs virtual landscapes from the ground up. Teleological algorithms do not construct, but rather destruct virtual environments by simulating rainfall and erosion. This inverse nature between the two algorithms produces interesting questions: could it be possible that the pseudo-randomness of the ontogenetic algorithm produces more convincing results than the more realistic process of the teleological algorithm? How difficult is it to mimic real world processes in a simulated environment? These questions arise because of how different the approaches of these two algorithms are, and thus reinforce the importance of examining the two in particular.

The second reason that these two algorithms are being examined as opposed to others is that they are frequently used as a baseline of which other more complex algorithms are built off of. For example, there exists an algorithm for procedurally generating terrain that makes use of software agents. A software agent is a function that can utilize a variety of functions in order to generate convincing land. This agent-based design involves many agents that each perform different tasks. Certain agents execute functions pseudo-randomly in order to generate a heightmap, which is an ontogenetic process. Other agents execute functions that are teleological and are used to modify the previously generated heightmap. Still others use a combination of ontogenetic, teleological, and other more complex processes to further modify the land [9]. Examining how these two basic algorithms work in turn allows for a deeper understanding of more involved algorithms such as this agent-based approach.

## 3.2: Ontogenetic Algorithm using Smooth Noise

The type of noise that this paper will be utilizing is an original implementation called smooth noise. Referring back to the description found in section **2.3.1**, noise is a function that, in this context, takes a point in two-dimensional space and produces a random number. However, generating completely random points and applying these as height values to a terrain mesh will lead to terrain that looks unnatural, as can be seen in *Example 4*. This is because in the real world, terrain is not truly random. Points that are near each other tend to have similar heights. This concept is known as *cohesion*. This term is an important concept in noise functions, as a noise function that generates more cohesive values also generates terrain that looks more natural.



Example 4: The mesh after mapping random heights to its vertices

Smooth noise attempts to be more cohesive by factoring in positions that are close to one another on the heightmap when performing its calculation. It achieves this by using a *lattice*, which can be thought of as an additional grid that is overlaid on top of the heightmap [10]. A visualization of a 3x3 lattice can be seen in *Example 5*.

A screen shot of a large window

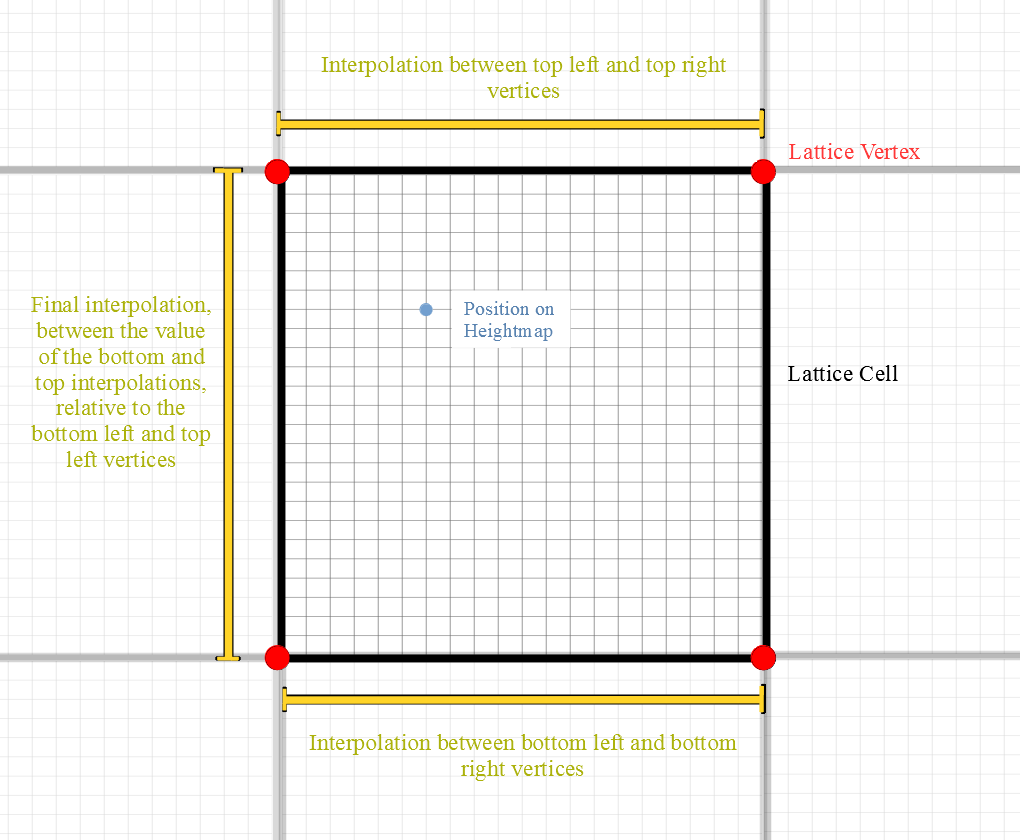
Description automatically generated

Example 5: A 3x3 lattice overlaying a heightmap

### 3.2.1: Algorithm

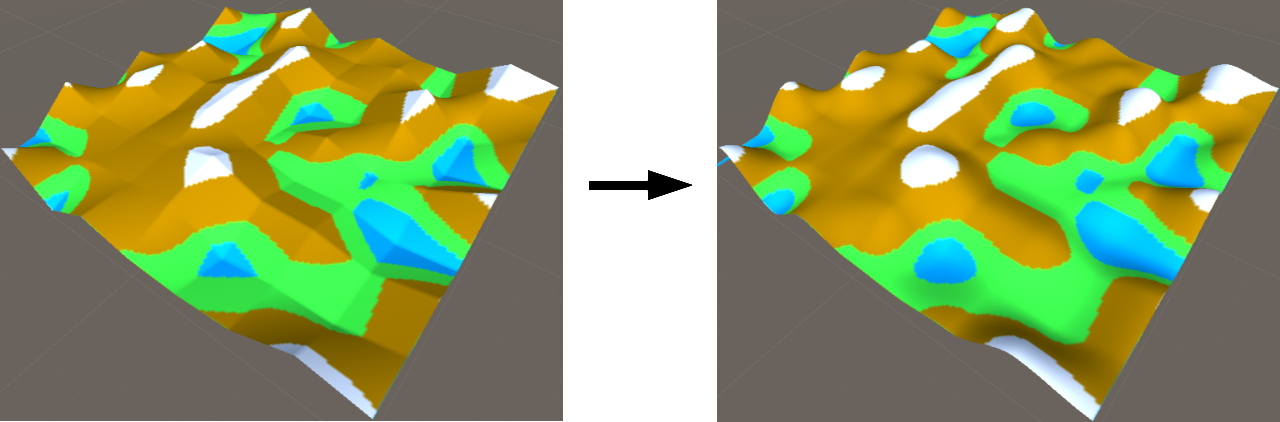
The smooth noise algorithm in question will now be explained step by step, with a visualization of important components of the algorithm provided in *Example 6*.

1. A flat heightmap of a particular size is generated
2. An integer lattice is laid over this heightmap
3. Every position on the heightmap is iterated through, where for every position, it:
   1. Finds the closest four lattice vertices that surround the current heightmap position
   2. Generates a pseudo-random height value for each of these four lattice vertices
   3. Uses a smoothing function to map the current heightmap position’s x and y values to smooth position values
   4. Linearly interpolates between the top left and top right lattice vertex values, with the smooth position’s x coordinate as the interpolation value
   5. Linearly interpolates between the bottom left and bottom right lattice vertex values, with the smooth position’s x coordinate as the interpolation value again
   6. Linearly interpolates from the value generated in step *3e* towards the value generated in step *3d*, with the smooth position’s y coordinate as the interpolation value
   7. Returns value generated in *3f* as the height value for the current position



Example 6: A cell of the lattice, zoomed in to show heightmap and vertices

This algorithm is called “smooth noise” because it utilizes a built-in Unity smoothing function called Mathf.SmoothStep() on the input position in step *3c* of the algorithm. Performing this smoothing further improves how natural the terrain looks when the heightmap is applied to the mesh, as can be seen in *Example 7*. This custom algorithm can be thought of as a simplified Perlin noise function that is modified with a smoothing step.



Example 7: The difference between terrain generated without the smoothing step and with the smoothing step

In the original Perlin noise function, the input is a two-dimensional coordinate in space just like the smooth noise function. However, where these two functions differ is between their lattice vertex calculations. Perlin noise utilizes gradient vectors with random directions that are generated at each lattice vertex surrounding the input position. It also includes four more vectors that point from each of the lattice vertices to the input position. The function then involves taking the dot product of these four vectors and the four random gradient vectors to be used in the interpolation step of the algorithm [6, 10].

Smooth noise simplifies this entire process by simply generating a random value rather than a random gradient vector at each of the four vertices. It also forgoes generating four vectors from each of the vertices to point to the input position, as there are no gradient vectors for the dot product calculation in this algorithm.

## 3.3: Teleological Algorithm using Droplet Erosion

The type of teleological algorithm that this project will utilize is a hydraulic erosion algorithm, simulating rainfall to modify previously generated terrain. Although it may seem ideal to model multiple droplets at the same time to better mimic how rain falls in the real world, this algorithm performs its calculation using one droplet at a time. In addition to this, the virtual droplet never leaves the ground and is thus two dimensional in nature. The reason for this is because the water droplet will be modifying a heightmap, which is a two-dimensional structure. This algorithm is much more complex than the ontogenetic algorithm, so having these restrictions helps to improve the ease of implementation and the processing speed.

### 3.3.1: Understanding Properties of the Droplet

It was noted in Section **2.4** that the water droplet is capable of two actions: eroding soil and depositing soil. With this information in mind, several elements that comprise the water droplet will be touched upon before walking through the algorithm step by step. These elements include a droplet’s *lifetime, size, speed,* and *soil capacity* [11]*.*

* *Lifetime* refers to how many times the droplet performs its actions before a new water droplet is generated. The actions that the droplet performs during its lifetime will be detailed in Section **3.3.2**
* *Size* refers to how many units of space that the droplet takes up. This influences what points on the heightmap the water droplet can erode soil from or deposit soil to
* *Soil Capacity* determines how much soil that the droplet is capable of carrying with it at any given moment. It is determined by combining the *Size* and *Speed* components. The larger the values of *Size* and *Speed*, the more soil the droplet can hold
* *Speed* is determined by the shape of the land that the droplet is traveling over. It is calculated based on the direction that the water droplet is moving on the heightmap, and whether the point that it is traveling to is lower than the current height. Note that *Speed* does not influence how far the droplet travels every time it moves - it is simply a factor for determining *Soil* *Capacity*.

### 3.3.2: Algorithm

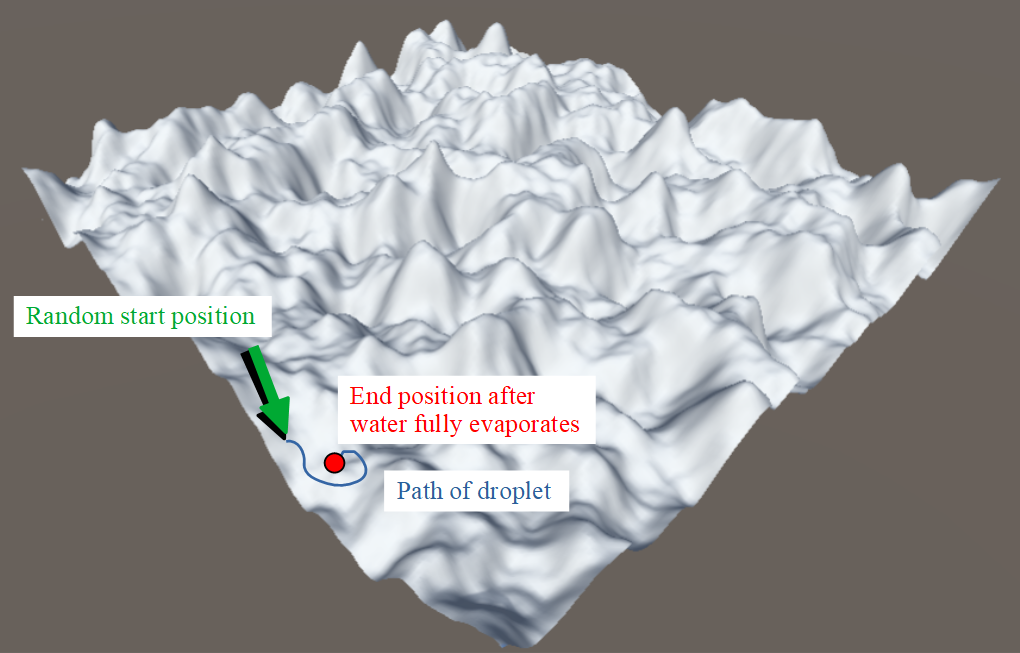
The droplet erosion algorithm will now be explained step by step, with a visualization of important components of the algorithm provided in *Example 8*.

1. A heightmap is passed to the droplet erosion algorithm for modification
2. It is established how many times the erosion function will run. For every iteration:
   1. A water droplet is created at a random point on the heightmap
   2. A lifetime is established. While the droplet is active, it:
      1. Determines its height on heightmap
      2. Determines a direction to move
      3. Moves one unit in the determined direction
      4. Determines its height at the new position that was just moved to
      5. Determines its soil capacity (high when moving fast down slope or droplet size is large)
      6. Determines if it is flowing up a slope or holding too much soil:

If true: Deposit soil to surrounding area

If false: Erode surrounding soil based on its size

* + 1. Updates its speed
    2. Evaporates a fraction of its water, decreasing its size

A picture containing map, text

Description automatically generated

Example 8: A mesh generated with the smooth noise algorithm, shown in these two images without color for clarity. A hypothetical path for a water droplet to take on any give iteration is illustrated in both images.

# 4: Algorithm Combination Comparison

This section will present several different combinations of ontogenetic and teleological algorithms, and each will be examined in terms of four properties. These properties include realism, processing power used during generation, ease of implementation, and the applicability this particular combination of algorithms might have in a video game world. A picture of the mesh, as well as a diagram detailing the combination of algorithms will be included at the top of every algorithm’s comparison section for clarity.

## 4.1: Random Noise

A close up of a coral

Description automatically generatedA screenshot of a cell phone

Description automatically generated

**Realism:** As explained in **Section** **3.2**, the terrain generated with random noise does not remotely resemble terrain found in the real world. This is because the terrain lacks cohesion. There are no lake-like or mountainous structures to be seen, and the land is covered in sharp spikes upward and sudden dips downward.

**Processing Power:** 9 milliseconds (0.0096068 seconds)

**Ease of Implementation:** The only process that this particular algorithm undergoes is generating pseudo-random value for every vertex on the heightmap. This is done trivially with a seeded noise function that is documented in **Section** **6.5.2.**

**Applicability:** Although it may seem to be completely unusable, this rugged terrain may have its unconventional uses. It seems to be ideal for creating a surface that would have a lot of friction, as anything rolling over such a bumpy surface would undoubtedly be slowed much faster than if the land were more smooth. Imagine a video game where the player controls a car, and a designer wishes to procedurally generate land for this car to travel across. Suppose this designer wanted to create a more difficult section that had patches of land that slowed the player’s car down. They very well might use a random noise function to generate jagged and uneven terrain to effectively achieve the generation of these patches of land, especially with how little processing power the algorithm requires to generate.

## 4.2: Smooth Noise

A close up of a colorful background

Description automatically generatedA picture containing screenshot

Description automatically generated

**Realism:** Because the smooth noise algorithm accounts for vertices on the mesh that are close to one another, the resulting terrain is more cohesive than random noise. There are gradual slopes and structures that resemble mountains and lakes. However, there are not many areas that are flat and grassy, which is quite unrealistic. In addition to this, because the smooth noise algorithm is only run for one iteration in this particular combination of algorithms, there is not much detail in the land that is generated as a whole. The land looks exaggerated, cartoon-like, and much like its namesake, much too smooth to be considered very realistic.

**Processing Power:** 74 milliseconds (0.07446604 seconds)

**Ease of Implementation:** The challenge when implementing the smooth noise algorithm came not from the logic behind the algorithm itself, but rather the amount of information about the structure of ontogenetic algorithms that was required to be learned beforehand. The algorithm had three main components: determining the current local position in space and obtaining information about the surrounding cell and its vertices, applying a smoothing function to this local position, and finally using interpolation to create cohesive height values.

Once the concept of a heightmap and a lattice were thoroughly understood, it was not particularly difficult to determine the current local position and be able to have a reference to the surrounding cell’s vertices. It was also not difficult to implement the smoothing function, as it was a built-in function (as mentioned in **Section 3.2.1**). However, it was quite challenging to implement the interpolation. This is because the idea was directly influenced by Ken Perlin’s implementation of Perlin noise, and thus a deep understanding of how that algorithm functioned was required before implementation.

**Applicability:** Although the land generated by smooth noise is not realistic, that does not mean it does not potentially have its uses. Given that the terrain has a smooth, cartoon look to it, it is not hard to imagine a children’s game or a game with a more cartoon-like art style utilizing these simplistic mountains. Accounting for the low amount of processing power required to generate this land, it also would not be difficult to imagine this type of algorithm being used on video game projects that have significant money or processing power limitations, or projects whose focus is on other aspects of the game other than the visuals.

## 4.3: Layered Smooth Noise

A picture containing food, colorful

Description automatically generatedA picture containing screenshot

Description automatically generated

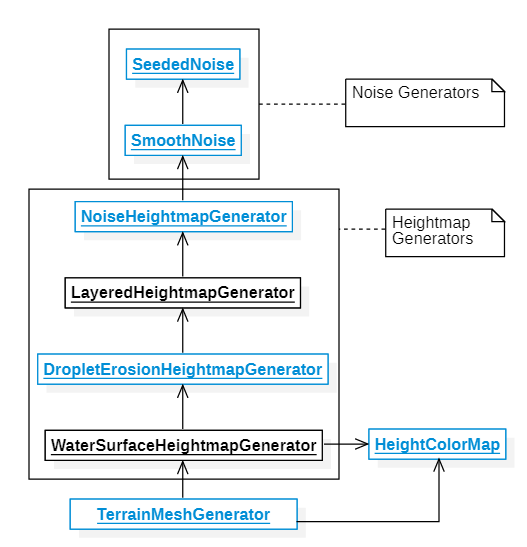
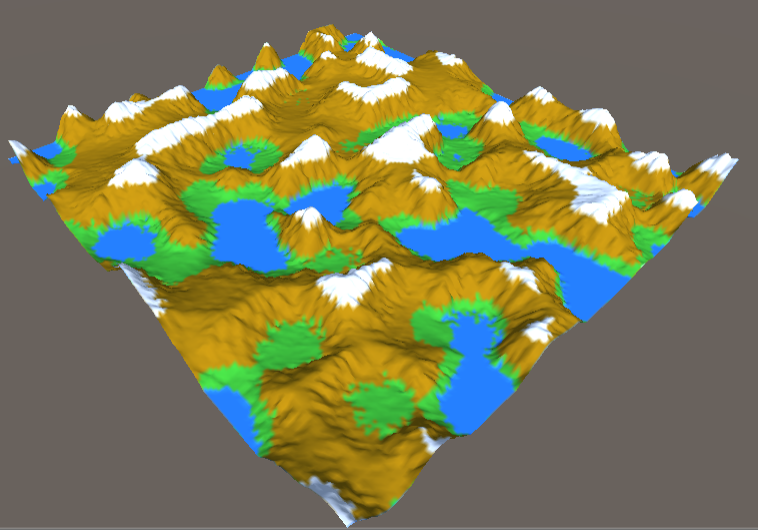
**Realism:** Compared to smooth noise, the structures generated are considerably more realistic. Layering multiple smooth noise algorithms allows for more fine detail on every iteration, and thus there is more variety in the land that is generated as a whole. Mountains are varied in their size and shape, with less of a smooth, artificial look like the smooth noise algorithm without layering. The mountains also have more different shapes, with some having sharp peaks and others having more rounded ones. There are more flat grassy areas, and there are many different kinds of shapes for the bodies of water that are generated.

**Processing Power:** 121 milliseconds (0.1216831seconds)

**Ease of Implementation:** This algorithm was not particularly difficult to implement once the idea of algorithmic layering was understood. The basic idea was to run a heightmap generator multiple times, with each iteration’s height values being added onto the previous iteration’s height values. The most difficult component of this algorithm was ensuring that the height and size were reduced every iteration, as this added more subtle detail to the previous iteration’s generated heightmap.

**Applicability:** This algorithm is very much the best of both worlds in terms of realism and processing power. The land generated is passably realistic, and the amount of processing power required to generate that land is minimal. It is a vast improvement over the realism of the previous smooth noise algorithm, and only requires around 47 more milliseconds to generate. The processing power required to generate land using this algorithm is nearly a fourth of the processing power needed to generate the droplet erosion algorithm, and could be applicable in a variety of different types of games if subtle detail is a non-issue.

## 4.4: Smooth Noise with Droplet Erosion



**Realism:** The step up in realism from the layered smooth noise algorithm to the current combination of algorithms is astronomical. Every part of the land that is generated is more realistic after running the droplet erosion simulation, because it adds more subtle details than any of the other combinations mentioned thus far. The mountains have ridges that mimic the ones found in nature, as well as numerous variations in their shape. Instead of mountains being placed individually around the land as seen in the land generated by previous algorithms, structures that resemble real world mountain ranges are present. There are also more flat grassland areas, and rounded, detailed bodies of water.

**Processing Power:** 458 milliseconds (0.4586088 seconds)

**Ease of Implementation:** The droplet erosion algorithm was significantly more difficult to implement than the smooth noise or layered algorithms for the sole reason that there were many more components to account for. Not only was it necessary to thoroughly understand how to manipulate and reference the vertices of the heightmap, understanding how to model the physics behind the water droplet was also required. In addition to setting up properties of the simulation space such as the inertia and gravity to be used, the water droplet required many properties to be manipulated. Some such properties included the speed at which the droplet evaporated, the amount of water it was holding, the amount that it was capable of eroding, and numerous others which are documented in **Section 6.3**. Having so many different components that all interacted with one another made programming and debugging this algorithm significantly more difficult, and thus led to a significantly longer implementation time when compared to the other algorithms examined.

**Applicability:** This combination of algorithms is much more realistic than any of the previous combinations that were examined, and would be best utilized in video games that have a more realistic art style. It would also be useful in areas outside the realm of video games, such as software that looks to simulate nature, as the method used to generate the terrain mimics natural processes more than previous algorithm combinations. The only downside that this combination of algorithms has in terms of its applicability is its significant increase in processing power. It requires nearly four times the amount of time to generate when compared to the layered smooth noise, for example, and this aspect of the algorithm would need to be thoroughly accounted for before being used by game developers.

## 4.5: Layered Smooth Noise with Droplet Erosion

A picture containing colorful, kite

Description automatically generatedA picture containing screenshot

Description automatically generated

**Realism:** This combination of algorithms is the most realistic out of any of the other combinations of algorithms that were examined in this research paper. There are simply more subtleties in the terrain because of the layered smooth noise contributing larger details (such as structures resembling mountain ranges and flat grasslands) and the droplet erosion contributing smaller details (such as the ridges on the mountains). Small pockets of water are present throughout the terrain, and the mountains also exhibit more varied shapes and structures. The areas of grass in the generated terrain are much more natural, with wider patches stretching between mountains and subtle slopes scattered throughout. The combination of all of these details leads the terrain generated by layered smooth noise with droplet erosion to be the most realistic.

**Processing Power:** 530 milliseconds (0.5301198 seconds)

**Ease of Implementation:** Because of the way that the structure for this project was set up, implementing this combination of algorithms was trivial, because the droplet erosion and layered heightmap algorithms were already created and functional. As can be seen by the diagrams provided alongside the pictures of the mesh in this section, this project was implemented in such a way that different components of the overall “pipeline” could be switched on and off to provide different combinations of algorithms to function together and produce terrain. In this particular combination of algorithms, for example, generating the layered heightmap generator with droplet erosion was as simple as activating the layered heightmap generator function and droplet erosion function.

**Applicability:** This combination of algorithms generates land that is not only more highly detailed than the previous combination, but with more flat grasslands and less water. Having more flat areas of grass could be utilized by game developers in a game that is more focused on player traversal, for example, as having flatter areas to walk in the game world and having less obstacles (such as water) would lead to more area for the player to be able to move around in. However, the processing power required to generate land using this particular combination of algorithms would have to be considered if a game were to be developed using it. This is because it takes the longest amount of time to generate when compared to the other combinations of algorithms examined in this project, and thus its applicability would depend entirely on the type of hardware that a game using this combination was being developed for.

# 5: Conclusion and Future Work

## 5.1: Choosing the Appropriate Procedural Terrain Generation Algorithm

It can be clearly seen from the previous section that there is no single algorithm or combination of algorithms that may be considered superior to others. For example, while the random noise algorithm generates land significantly faster and is easier to implement than all of the other combinations of algorithms, the land that it generates does not contain any features that allow it to remotely resemble real-world terrain. On the other hand, the combination of the layered smooth noise algorithm with the droplet erosion algorithm provides the most realistic terrain, but the implementation of the droplet erosion algorithm is significantly more difficult, and the processing power is much higher than the other combinations of algorithms.

The fact that there is not one perfect solution to procedural terrain generation is not a disadvantage when it comes to utilizing these algorithms, however. Modern video games are developed by many different types of studios, and the developers at these studios have entirely different needs. Many independent developers develop games with miniscule budgets and still other developers are part of multimillion-dollar companies and develop video games for the latest generation of video game hardware. Choosing what type of procedural terrain generation would best fit a particular game (or even choosing whether or not to use procedural terrain generation in the first place) largely depends on the type of game that is being developed and the amount of resources that are available.

As is the case with most aspects of game development, it is wisest for game developers to take advantage of the tools that will most benefit the type of project that is being worked on. Being able to understand the differences between different algorithms and combinations of algorithms for procedural terrain generation makes for a more informed game developer, and one that is better capable of developing a game that can truly take advantage of these powerful tools.

## 5.2: Future Work

The goal of this project was to provide more information to those interested in learning about the benefit of procedural generation in video games and how two important procedural terrain generation algorithms functioned. For this reason, the algorithms and combinations of algorithms presented in the paper were implemented with the goal of providing better visualizations and useful information regarding their inherent advantages and disadvantages. They were not implemented in such a way that they could be used in a video game, at least not with some major modifications.

Several ways that the algorithms could be improved will now be discussed briefly. A step to improve the base algorithms was already taken with the implementation of the water surface heightmap generator. As can be seen in the implementation diagrams found throughout **Section 4**, the water surface heightmap generator is the last heightmap generator in the implementation pipeline that all of the algorithm combinations utilize before being inputted into the terrain mesh generator. This additional step improves the appearance of the generated land by producing bodies of water that have depth, rather than simply being surface terrain that is just colored to look like water.

As shown in the images provided throughout **Section 4**, there are areas of grass that appear on the sides of mountains and steep terrain, which make the generated terrain look less realistic than if grass were to only appear in flatter areas. If work were to continue on these algorithms, implementing an additional heightmap generator to prevent this grass from appearing on steep slopes that occurred after the water surface heightmap generator in the pipeline would be ideal.

Another modification to this implementation that would greatly improve the realism of the terrain generated would be an ability to use textures with the terrain mesh generator rather than simple colors. The current implementation uses four colors to represent aspects of terrain, which include blue for water, green for grass, brown for mountains, and white for snow. While these colors are helpful for visualizing the algorithms, being able to map realistic looking water, grass, mountain, and snow textures onto the generated terrain would improve the realism significantly.

# 6: Appendix

## 6.1: Heightmap Generator

**public** **abstract** class HeightmapGenerator **:** MonoBehaviour **{**

**public** **abstract** float**[,]** GenerateHeightmap**(**Vector2Int size**,** float scale**);**

**}**

## 6.2: Noise Generator

**public** **abstract** class NoiseGenerator **:** MonoBehaviour **{**

**public** **abstract** float Generate **(**Vector2 position**);**

**}**

## 6.3: Droplet Erosion

**public** class DropletErosion **:** MonoBehaviour **{**

**public** int iterations**;**

**public** float minSlope**;**

**public** float gravity**;**

**[**Range**(**0**,** 1**)]**

**public** float inertia**;**

**public** int lifetime**;**

**public** float capacity**;**

**[**Range**(**0**,** 1**)]**

**public** float evaporationSpeed**;**

**public** float erosionPercent**;**

**public** float depositionPercent**;**

**public** float initialWater**;**

**public** float initialSpeed**;**

**private** float initialSedimentCarried**;**

**public** void Erode**(**float**[,]** heightmap**)** **{**

// GetLength(0) refers to x length, heightmap.GetLength(1) refers to y length

Vector2 heightmapSize **=** **new** Vector2**(**

heightmap**.**GetLength**(**0**),**

heightmap**.**GetLength**(**1**));**

**for** **(**int d **=** 0**;** d **<** iterations**;** d**++)** **{**

// Pick a random spot for the droplet to spawn in

Vector2 position **=** **new** Vector2**(**

UnityEngine**.**Random**.**Range**(**0**,** heightmapSize**.**x**),**

UnityEngine**.**Random**.**Range**(**0**,** heightmapSize**.**y**));**

SimulateDroplet**(**heightmap**,** position**);**

**}**

**}**

**private** void SimulateDroplet**(**float**[,]** heightmap**,** Vector2 position**)** **{**

Vector2 heightmapSize **=** **new** Vector2**(**

heightmap**.**GetLength**(**0**),** // get length of x of heightmap

heightmap**.**GetLength**(**1**));** // get length of y of heightmap

Vector2 direction **=** Vector2**.**zero**;**

float water **=** initialWater**;**

float speed **=** initialSpeed**;**

float sedimentCarried **=** initialSedimentCarried**;**

**for** **(**int l **=** 0**;** l **<** lifetime**;** l**++)** **{**

// If droplet outside of map, then quit current simulation

**if** **(**position**.**x **<** 0 **||**

position**.**y **<** 0 **||**

position**.**x **+** 1 **>=** heightmapSize**.**x **||**

position**.**y **+** 1 **>=** heightmapSize**.**y**)**

**return;**

// Calculate height

**(**float height**,** Vector2 gradient**)** **=** CalculateHeightAndGradient**(**heightmap**,** position**);**

// Calculate new direction

direction **=** Vector2**.**Lerp**(-**gradient**,** direction**,** inertia**).**normalized**;**

// Move one unit in determined direction

Vector2 lastPosition **=** position**;**

position **+=** direction**;**

// If droplet outside of map, then quit current simulation

**if** **(**position**.**x **<** 0 **||** position**.**y **<** 0 **||**

position**.**x **+** 1 **>=** heightmapSize**.**x **||** position**.**y **+** 1 **>=** heightmapSize**.**y **||**

direction**.**magnitude **<** float**.**Epsilon**)**

**return;**

// Determine height at new position

// Note: "\_" denotes that the gradient is discarded in returned tuple

**(**float newHeight**,** \_**)** **=** CalculateHeightAndGradient**(**heightmap**,** position**);**

// Determiine what happens when droplet moves downhill

float heightDiff **=** newHeight **-** height**;**

bool dropletMovingUphill **=** heightDiff **>** 0**;**

float carryCapacity **=** Mathf**.**Max**(-**heightDiff**,** minSlope**)** **\*** speed **\*** water **\*** capacity**;**

// Deposit or erode depending on whether droplet is moving uphill

**if** **(**dropletMovingUphill **||** sedimentCarried **>** carryCapacity**)** **{**

//deposit

//Spike avoidance by depositing less when moving uphill

float depositAmount **=** dropletMovingUphill **?**

Mathf**.**Min**(**heightDiff**,** sedimentCarried**)** **:**

**(**sedimentCarried **-** carryCapacity**)** **\*** depositionPercent**;**

ChangeHeight**(**heightmap**,** lastPosition**,** depositAmount**);**

sedimentCarried **-=** depositAmount**;**

**}** **else** **{**

//erode

float erodeAmount **=** Mathf**.**Min**(**

**(**carryCapacity **-** sedimentCarried**)** **\*** erosionPercent**,** **-**heightDiff**);**

ChangeHeight**(**heightmap**,** lastPosition**,** **-**erodeAmount**);**

sedimentCarried **+=** erodeAmount**;**

**}**

// Update speed based on steepness of hill and gravity of environment

speed **=** Mathf**.**Sqrt**(**Mathf**.**Max**(**0**,** **(**speed **\*** speed**)** **+** heightDiff **\*** gravity**));**

// Evaporate water

water **\*=** **(**1 **-** evaporationSpeed**);**

**}**

**}**

**private** void ChangeHeight**(**float**[,]** heightmap**,** Vector2 position**,** float heightChange**){**

Vector2Int cellPosition **=** **new** Vector2Int**(**

**(**int**)**position**.**x**,**

**(**int**)**position**.**y**);**

Vector2 localPosition **=** **new** Vector2**(**

position**.**x **-** **(**int**)**position**.**x**,**

position**.**y **-** **(**int**)**position**.**y**);**

float lerpL **=** 1 **-** localPosition**.**x**;**

float lerpR **=** localPosition**.**x**;**

float lerpT **=** localPosition**.**y**;**

float lerpB **=** 1 **-** localPosition**.**y**;**

float lerpBL **=** lerpB **\*** lerpL**;**

float lerpBR **=** lerpB **\*** lerpR**;**

float lerpTL **=** lerpT **\*** lerpL**;**

float lerpTR **=** lerpT **\*** lerpR**;**

heightmap**[**cellPosition**.**x **+** 0**,** cellPosition**.**y **+** 1**]** **+=** heightChange **\*** lerpTL**;**

heightmap**[**cellPosition**.**x **+** 1**,** cellPosition**.**y **+** 1**]** **+=** heightChange **\*** lerpTR**;**

heightmap**[**cellPosition**.**x **+** 0**,** cellPosition**.**y **+** 0**]** **+=** heightChange **\*** lerpBL**;**

heightmap**[**cellPosition**.**x **+** 1**,** cellPosition**.**y **+** 0**]** **+=** heightChange **\*** lerpBR**;**

**}**

// Returns a tuple

**private** **(**float**,** Vector2**)** CalculateHeightAndGradient**(**float**[,]** heightmap**,** Vector2 position**){**

// Height calculation

Vector2Int cellPosition **=** **new** Vector2Int**(**

**(**int**)**position**.**x**,**

**(**int**)**position**.**y**);**

Vector2 localPosition **=** **new** Vector2**(**

position**.**x **-** **(**int**)**position**.**x**,**

position**.**y **-** **(**int**)**position**.**y**);**

float heightTL **=** heightmap**[**cellPosition**.**x **+** 0**,** cellPosition**.**y **+** 1**];**

float heightTR **=** heightmap**[**cellPosition**.**x **+** 1**,** cellPosition**.**y **+** 1**];**

float heightBL **=** heightmap**[**cellPosition**.**x **+** 0**,** cellPosition**.**y **+** 0**];**

float heightBR **=** heightmap**[**cellPosition**.**x **+** 1**,** cellPosition**.**y **+** 0**];**

float lerpTop **=** Mathf**.**Lerp**(**heightTL**,** heightTR**,** localPosition**.**x**);**

float lerpBottom **=** Mathf**.**Lerp**(**heightBL**,** heightBR**,** localPosition**.**x**);**

float height **=** Mathf**.**Lerp**(**lerpBottom**,** lerpTop**,** localPosition**.**y**);**

// Gradient calculation [11]

Vector2 gradient **=** **new** Vector2**(**

**(**heightBR **-** heightBL**)** **\*** **(**1 **-** localPosition**.**y**)** **+** **(**heightTR **-** heightTL**)** **\*** localPosition**.**y**,**

**(**heightTL **-** heightBL**)** **\*** **(**1 **-** localPosition**.**x**)** **+** **(**heightTR **-** heightBR**)** **\*** localPosition**.**x**);**

**return** **(**height**,** gradient**);**

**}**

**}**

### 6.3.1: Droplet Erosion Heightmap Generator

**public** class DropletErosionHeightmapGenerator **:** HeightmapGenerator **{**

**public** HeightmapGenerator heightmapGenerator**;**

**private** DropletErosion dropletErosion**;**

**private** float**[,]** heightmap**;**

**private** void Start**()** **{**

dropletErosion **=** GetComponent**<**DropletErosion**>();**

**}**

**public** **override** float**[,]** GenerateHeightmap**(**Vector2Int size**,** float scale**)** **{**

heightmap **=** **(**heightmap **!=** **null)** **?** heightmap **:**

heightmapGenerator**.**GenerateHeightmap**(**size**,** scale**);**

dropletErosion**.**Erode**(**heightmap**);**

**return** heightmap**;**

**}**

**}**

## 6.4: Layer Generator

**public** class LayeredHeightmapGenerator **:** HeightmapGenerator **{**

**public** HeightmapGenerator heightmapGenerator**;**

**public** int iterations**;** // "octaves"

**[**Range**(**0**,**1**)]** **public** float sizeScalePerIteration**;**

**[**Range**(**0**,**1**)]** **public** float heightScalePerIteration**;**

**public** **override** float**[,]** GenerateHeightmap**(**Vector2Int size**,** float scale**)** **{**

float**[,]** heightmap **=** **new** float**[**size**.**x**,** size**.**y**];**

// Let SSPI = 0.5. If i is 0, scale = 1

// If i is 1, scale = 0.5

// If i is 2, scale = 0.25...

**for** **(**int i **=** 0**;** i **<** iterations**;** i**++)** **{**

float heightScale **=** Mathf**.**Pow**(**heightScalePerIteration**,** i**);**

float sizeScale **=** scale **\*** Mathf**.**Pow**(**sizeScalePerIteration**,** i**);**

float**[,]** layer **=** heightmapGenerator**.**GenerateHeightmap**(**size**,** sizeScale**);**

**for** **(**int xi **=** 0**;** xi **<** size**.**x**;** xi**++)**

**for** **(**int yi **=** 0**;** yi **<** size**.**y**;** yi**++)**

heightmap**[**xi**,** yi**]** **+=** layer**[**xi**,** yi**]** **\*** heightScale**;**

**}**

**return** heightmap**;**

**}**

**}**

## 6.5: Noise Generators

### 6.5.1: Noise Heightmap Generator

**public** class NoiseHeightmapGenerator **:** HeightmapGenerator **{**

**public** Vector2 offset**;**

**public** NoiseGenerator sourceNoise**;**

**public** **override** float**[,]** GenerateHeightmap**(**Vector2Int size**,** float scale**)** **{**

float**[,]** heightMap **=** **new** float**[**size**.**x**,** size**.**y**];**

// Position in vertex space (on the heightmap) : (xi, yi)

**for** **(**int xi **=** 0**;** xi **<** size**.**x**;** xi**++)** **{**

**for** **(**int yi **=** 0**;** yi **<** size**.**y**;** yi**++)** **{**

// Scale variable translates vertex space into lattice space

// Position in lattice space (in the noise) : (position.x, position.y)

Vector2 position **=** **new** Vector2**(**xi**,** yi**)** **/** scale**;**

heightMap**[**xi**,** yi**]** **=** sourceNoise**.**Generate**(**position **+** offset**);**

**}**

**}**

**return** heightMap**;**

**}**

**}**

### 6.5.2: Seeded Noise

**public** class SeededNoise **:** NoiseGenerator **{**

// [12]

**public** **override** float Generate**(**Vector2 position**)** **{**

float f **=** Mathf**.**Sin**(**Vector2**.**Dot**(**position**,** **new** Vector2**(**12.9898f**,** 78.233f**)))** **\*** 43758.5453f**;**

**return** Mathf**.**Abs**(**f **-** **(**int**)**f**);**

**}**

**}**

### 6.5.3: Smooth Noise

**public** class SmoothNoise **:** NoiseGenerator **{**

**private** static Dictionary**<**Vector2Int**,** Vector4**>** cellHeightCache **=**

**new** Dictionary**<**Vector2Int**,** Vector4**>();**

**public** NoiseGenerator sourceNoise**;**

**public** **override** float Generate**(**Vector2 position**){**

Vector4 cellHeights **=** GetOrGenerateHeights**(**position**);**

float heightTL **=** cellHeights**.**x**;**

float heightTR **=** cellHeights**.**y**;**

float heightBL **=** cellHeights**.**z**;**

float heightBR **=** cellHeights**.**w**;**

Vector2 cellBL **=** **new** Vector2**(**Mathf**.**Floor**(**position**.**x**),** Mathf**.**Floor**(**position**.**y**));**

// Calculating local position in space

Vector2 localPosition **=** position **-** cellBL**;**

// Smoothing out local position

float smoothX **=** Mathf**.**SmoothStep**(**0**,** 1**,** localPosition**.**x**);**

float smoothY **=** Mathf**.**SmoothStep**(**0**,** 1**,** localPosition**.**y**);**

Vector2 smoothLocalPosition **=** **new** Vector2**(**smoothX**,** smoothY**);**

float lerpTop **=** Mathf**.**Lerp**(**heightTL**,** heightTR**,** smoothLocalPosition**.**x**);**

float lerpBottom **=** Mathf**.**Lerp**(**heightBL**,** heightBR**,** smoothLocalPosition**.**x**);**

**return** Mathf**.**Lerp**(**lerpBottom**,** lerpTop**,** smoothLocalPosition**.**y**);**

**}**

**private** Vector4 GetOrGenerateHeights**(**Vector2 position**)** **{**

Vector2Int cell **=** **new** Vector2Int**((**int**)**position**.**x**,** **(**int**)**position**.**y**);**

**if** **(**cellHeightCache**.**TryGetValue**(**cell**,** **out** Vector4 heights**)){**

**return** heights**;**

**}** **else** **{**

// Creating reference to four corners surrounding input point

Vector2 cellTL **=** **new** Vector2**((**int**)**position**.**x **+** 0**,** **(**int**)**position**.**y **+** 1**);**

Vector2 cellTR **=** **new** Vector2**((**int**)**position**.**x **+** 1**,** **(**int**)**position**.**y **+** 1**);**

Vector2 cellBL **=** **new** Vector2**((**int**)**position**.**x **+** 0**,** **(**int**)**position**.**y **+** 0**);**

Vector2 cellBR **=** **new** Vector2**((**int**)**position**.**x **+** 1**,** **(**int**)**position**.**y **+** 0**);**

// Generating random values for all four corners around input point

float heightTL **=** sourceNoise**.**Generate**(**cellTL**);**

float heightTR **=** sourceNoise**.**Generate**(**cellTR**);**

float heightBL **=** sourceNoise**.**Generate**(**cellBL**);**

float heightBR **=** sourceNoise**.**Generate**(**cellBR**);**

heights **=** **new** Vector4**(**heightTL**,** heightTR**,** heightBL**,** heightBR**);**

cellHeightCache**[**cell**]** **=** heights**;**

**return** heights**;**

**}**

**}**

**}**

### 6.5.4: Water Surface

**public** class WaterSurfaceHeightmapGenerator **:** HeightmapGenerator **{**

**public** HeightColorMap heightColorMap**;**

**public** HeightmapGenerator heightmapGenerator**;**

**public** **override** float**[,]** GenerateHeightmap**(**Vector2Int size**,** float scale**)** **{**

float waterHeight **=** heightColorMap**.**zones**[**0**].**maxHeight**;**

float**[,]** heightmap **=** heightmapGenerator**.**GenerateHeightmap**(**size**,** scale**);**

**for** **(**int xi **=** 0**;** xi **<** size**.**x**;** xi**++)**

**for** **(**int yi **=** 0**;** yi **<** size**.**y**;** yi**++)**

**if** **(**heightmap**[**xi**,** yi**]** **<=** waterHeight**)**

heightmap**[**xi**,** yi**]** **=** waterHeight**;**

**return** heightmap**;**

**}**

**}**

## 6.6: Height Color Map

**public** class HeightColorMap **:** ScriptableObject **{**

**public** Zone**[]** zones**;**

**public** Color defaultZone**;**

**public** struct Zone **{**

**public** float maxHeight**;**

**public** Color color**;**

**}**

**public** Color CalculateZoneColor**(**float height**)** **{**

**foreach** **(**Zone zone **in** zones**)**

**if** **(**height **<=** zone**.**maxHeight**)**

**return** zone**.**color**;**

**return** defaultZone**;**

**}**

**}**

## 6.7: Terrain Mesh Generator

**public** class TerrainMeshGenerator **:** MonoBehaviour **{**

**public** Vector2Int size**;**

**public** float cellSpacing**;**

**public** float heightScale**;**

**public** float sizeScale **=** 1**;**

**public** HeightmapGenerator heightmapGenerator**;**

**public** HeightColorMap heightColorMap**;**

**public** bool continuousGeneration**;**

**private** Mesh mesh**;**

**private** MeshFilter meshFilter**;**

**private** MeshRenderer meshRenderer**;**

**private** Texture2D meshTexture**;**

**private** void Start**()** **{**

Assert**.**IsNotNull**(**heightmapGenerator**,** "Missing HeightmapGenerator"**);**

meshFilter **=** GetComponent**<**MeshFilter**>();**

meshRenderer **=** GetComponent**<**MeshRenderer**>();**

GenerateMesh**();**

GenerateTexture**();**

meshFilter**.**mesh **=** mesh**;**

**if** **(!**continuousGeneration**)**

ApplyHeightmap**();**

**}**

**private** void Update**()** **{**

**if** **(**continuousGeneration**)**

ApplyHeightmap**();**

**}**

// Changes every vertex in the mesh to match with heightmap

**private** void ApplyHeightmap**()** **{**

// Testing performance of generation

var stopwatch **=** **new** System**.**Diagnostics**.**Stopwatch**();**

stopwatch**.**Start**();**

float**[,]** heightmap **=** heightmapGenerator**.**GenerateHeightmap**(**

size **+** Vector2Int**.**one**,** sizeScale**);**

// Performance of generation console logging

stopwatch**.**Stop**();**

Debug**.**LogFormat**(**"Heightmap Generator {0} took {1} seconds"**,**

heightmapGenerator**,** stopwatch**.**Elapsed**);**

Debug**.**LogFormat**(**"Heightmap Generator {0} took {1} milliseconds"**,**

heightmapGenerator**,** stopwatch**.**ElapsedMilliseconds**);**

Vector3**[]** vertices **=** mesh**.**vertices**;**

Color**[]** colors **=** **new** Color**[**vertices**.**Length**];**

**for** **(**int xi **=** 0**;** xi **<=** size**.**x**;** xi**++){**

**for** **(**int yi **=** 0**;** yi **<=** size**.**y**;** yi**++)** **{**

int vi **=** VertexIndex**(**xi**,** yi**);**

float height **=** heightmap**[**xi**,** yi**];**

float scaledHeight **=** height **\*** heightScale**;**

vertices**[**vi**]** **=** **new** Vector3**(**vertices**[**vi**].**x**,** scaledHeight**,** vertices**[**vi**].**z**);**

colors**[**vi**]** **=** heightColorMap**.**CalculateZoneColor**(**height**);**

**}**

**}**

mesh**.**vertices **=** vertices**;**

mesh**.**RecalculateNormals**();**

mesh**.**RecalculateBounds**();**

meshTexture**.**SetPixels**(**colors**);**

meshTexture**.**Apply**();**

**}**

**private** void GenerateMesh**(){**

mesh **=** **new** Mesh**();**

Vector3**[]** vertices **=** **new** Vector3**[(**size**.**x **+** 1**)** **\*** **(**size**.**y **+** 1**)];**

Vector2**[]** uvMap **=** **new** Vector2**[**vertices**.**Length**];**

int**[]** triangles **=** **new** int**[**2 **\*** size**.**x **\*** size**.**y **\*** 3**];**

// Assigns each vertex a coordiante in the form (x, y)

**for** **(**int vx **=** 0**;** vx **<=** size**.**x**;** vx**++){**

**for** **(**int vy **=** 0**;** vy **<=** size**.**y**;** vy**++){**

vertices**[**VertexIndex**(**vx**,** vy**)]** **=** **new** Vector3**(**vx**,** 0**,** vy**)** **\*** cellSpacing**;**

uvMap**[**VertexIndex**(**vx**,**vy**)]** **=** **new** Vector2**((**float**)**vx **/** size**.**x**,** **(**float**)**vy **/** size**.**y**);**

**}**

**}**

// Assigns each cell 2 triangles, referencing 3 vertices (clockwise) each

int ti **=** 0**;**

**for** **(**int cx **=** 0**;** cx **<** size**.**x**;** cx**++){**

**for** **(**int cy **=** 0**;** cy **<** size**.**y**;** cy**++){**

triangles**[**ti**++]** **=** VertexIndex**(**cx **+** 0**,** cy **+** 0**);**

triangles**[**ti**++]** **=** VertexIndex**(**cx **+** 1**,** cy **+** 1**);**

triangles**[**ti**++]** **=** VertexIndex**(**cx **+** 1**,** cy **+** 0**);**

triangles**[**ti**++]** **=** VertexIndex**(**cx **+** 0**,** cy **+** 0**);**

triangles**[**ti**++]** **=** VertexIndex**(**cx **+** 0**,** cy **+** 1**);**

triangles**[**ti**++]** **=** VertexIndex**(**cx **+** 1**,** cy **+** 1**);**

**}**

**}**

mesh**.**vertices **=** vertices**;**

mesh**.**uv **=** uvMap**;**

mesh**.**triangles **=** triangles**;**

**}**

**private** void GenerateTexture**(){**

// +1 to size to account for there existing 1 more vertex than cells per axis

meshTexture **=** **new** Texture2D**(**size**.**x **+** 1**,** size**.**y **+** 1**,** TextureFormat**.**ARGB32**,** **false);**

meshRenderer**.**material**.**mainTexture **=** meshTexture**;**

**}**

// Converts a coordinate into its index. For use with converting 2D array -> 1D array

**private** int VertexIndex**(**int x**,** int y**)** **=>** y **\*** **(**size**.**x **+** 1**)** **+** x**;**

**}**

# 7: References

[1] M. Korth, “Everynes,” problemkaputt.de, 2004. [Online]. Available: problemkaputt.de/everynes.htm. [Accessed: Dec. 7, 2019].

[2] G. Christina, “Minecraft Unit Sales Worldwide 2016-2019,” statista.com, para. 1, Aug. 9, 2019. [Online]. Available: https://www.statista.com/statistics/680124/minecraft-unit-sales-worldwide/. [Accessed: Oct. 31, 2019].

[3] S. Oliveira and L. Magalhães, “Adaptive content generation for games,” In 24th Encontro Português de Computação Gráfica e Interação (EPCGI), 2017.

[4] J. Chan, J. J. Daza, W. Kwan, A. Basu. “Facilitating player progression by implementing procedural music in videogames,” In IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2017.

[5] T. J. Rose and A. G. Bakaoukas, “Algorithms and Approaches for Procedural Terrain Generation – A Brief Review of Current Techniques,” In 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES), 2016.

[6] T. Hyttinen, E. Mäkinen, T. Poranen, “Terrain Synthesis Using Noise by Examples,” In Proceedings of the 21st International Academic Mindtrek Conference ’17, 2017, pp. 17-25.

[7] H. Zhang, D. Qu, Y. Hou, F. Gao, F. Huang, “Synthetic Modeling Method for Large Scale Terrain Based on Hydrology,” IEEE Access, vol. 4, pp. 6238-6249, 2016.

[8] O. Št'ava, B. Beneš, M. Brisbin, J. Křivánek, “Interactive Terrain Modeling using Hydraulic Erosion,” in proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2008, pp. 201-210.

[9] J. Doran and I. Parberry, “Controlled Procedural Terrain Generation Using Software Agents,” IEEE Transactions on Computational Intelligence and AI in Games, vol. 2, no.2, 2010.

[10] K. Perlin, “An Image Synthesizer,” In SIGGRAPH ’85 Proceedings on the 12th annual conference on Computer graphics and interactive techniques, 1985, pp. 287-296.

[11] H. T. Beyer, “Implementation of a Method for Hydraulic Erosion,” B. Thesis, Technical University of Munich, Munich, Germany, 2015.

[12] U. Technologies, "Unity - Manual: Unity Manual", Docs.unity3d.com, 2015. [Online]. Available: http://docs.unity3d.com/Manual/index.html. [Accessed: Dec. 7, 2019].