Procedural Terrain Generation using Noise Function

DH2323 Computer Graphics and Interaction
Chloe Joyce Wabingga Raneses cjwr@kth.se

1 Introduction and Background

Creating 3D terrain and landscape can be done in different ways, for example by manually modeling in applications such as Blender. However, this method is inefficient if several different landscapes are needed in a short span of time. A faster, more efficient approach is to create a program that can generate the 3D terrain. This is the definition of procedural generation, to create content like worlds and map through algorithms rather than doing it manually one-by-one [1].

In terms of map generation, a common approach is to create a height map. A height map is a representation of elevation over a surface, most commonly used in computer graphics as a grayscale image where the brightness of each pixel represents a height value [2]. For terrain generation, the height map can be a simple array of values. However, just randomly filling this array with numbers is not enough, as this approach will create jagged terrain rather than the smooth landscape that is desired. The height map is filled with values generated through a noise function.

There are different types of noise, the simplest being white noise [3] . White noise is a random distribution of values, where each one is completely independent of its neighbor, resulting in sudden jumps of value and no smoothness in terrain. Other noise, such as value noise and Perlin noise, will try to eliminate this abrupt change using a blend of values from nearby points through sampling and interpolation. The choice of noise function will determine the overall appearance and smoothness of the generated terrain.

2 Implementation and Results

2.1 Noise function

The implemented noise function takes as input two numbers, x and y, and returns a float in the range of [0,1], representing from black (0) to white (1). The input values are used to create a vector (x, y), which is further used to obtain the integer and fractional coordinates and stored in two new vectors, **i** and **f**. Obtaining integer coordinates is done using the floor() function, and the fractional coordinates by taking the float minus its floor() value.

These two vectors are used to represent a 1x1 grid cell wherein the vector **i** makes up the bottom-left corner of the grid, and the fractional coordinates **f** represent the offset within the current cell. Note that the function does not actually create a grid, we just work as if there is one. On this 'grid', at each four corners, a random value is assigned. These four new values are then combined through interpolation, interpolating first horizontally, then vertically using the mix() function from GLM. As a result, the floats generated from this entire function can create an unpredictable but smooth pattern, in which nearby inputs produce similar output without major jumps in value [4].

Assigning random values to the four corners is what gives the noise function its unpredictability and variation, serving as anchor points for the terrain within that cell. Without randomizing these corners, every cell would be the same resulting in flat or repetitive terrain. More importantly, interpolating between these values is really the core of the noise function. Without interpolating, we would instead just get white noise, with random values at each grid point. Interpolation is what provides the continuity and smoothness of the terrain [5]. With such an impact on the results, how we interpolate also plays an important role. Pure linear interpolation will result in boxer, more sharp edges, especially where the slopes might change abruptly [6]. To fix this, we use a smoothstep function that helps in interpolating smoothly, more specifically, using the cubic polynomial [7]:

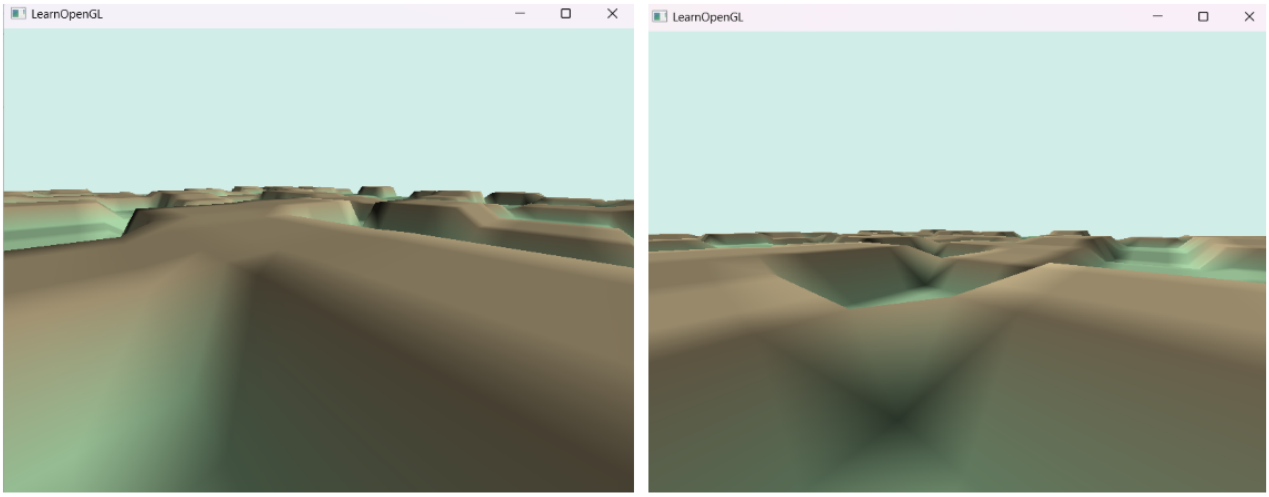

Figure 1: a) With smoothstep, b) without smoothstep

While the difference may be very subtle and mostly unnoticeable unless one is really looking for it, the smooth edges are still a desired trait and easy to apply.

2.2 Filling the height map

Implementing the noise function is not enough to create the terrain generator. Running it as it is would result in the following:

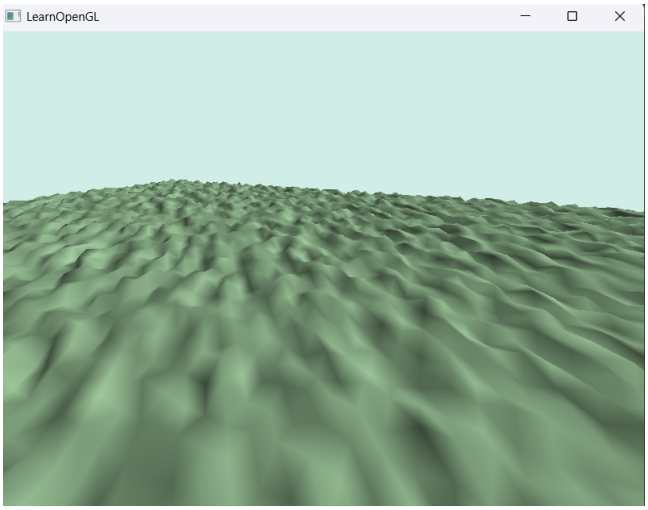

Figure 2: Terrain without using frequency and amplitude

To create a more dramatic landscape, we make use of frequency and amplitude. The frequency of noise can be likened to sine waves [8], and controls the horizontal scale of the hills. Thus, a lower frequency results in fewer but wider hills across the map, and a high frequency will create lots of little hills and valleys. To add frequency to our code, we simply multiply the input to the noise function by the frequency factor. Furthermore, there is also the amplitude factor, that we can multiply the output with. The amplitude controls the vertical size of the terrain. In other words, the height of hills and valleys. These two factors are used inside of the function in which the height map array is filled, on the line where the noise function call is made. The frequency and amplitude values can be played with to obtain different results, and finding the right balance is crucial.

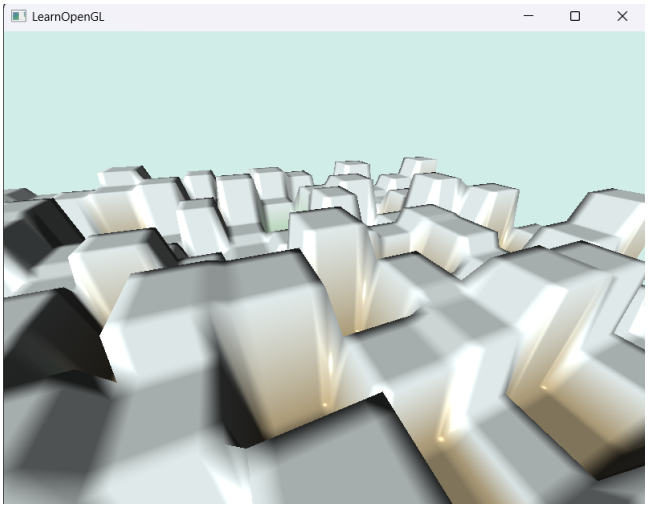

Figure 3: Frequency = 0.15, Amplitude = 15.0

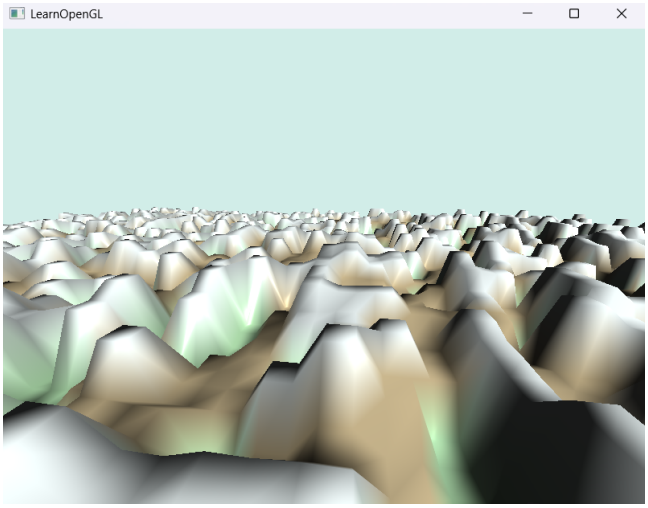

Figure 4: Frequency = 0.5, Amplitude = 5.0

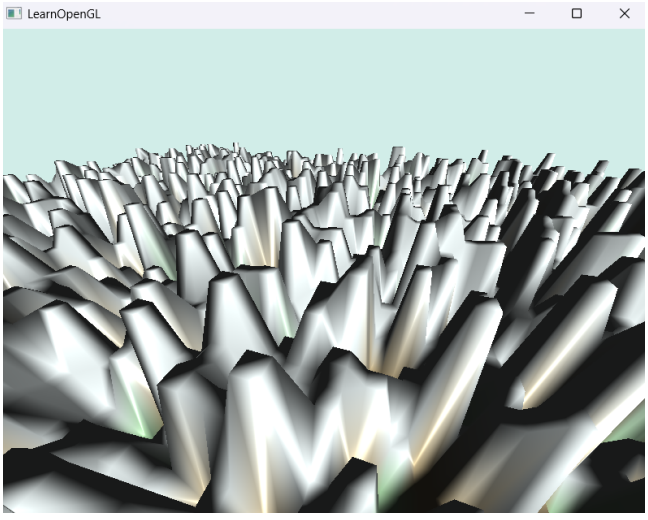

Figure 5: Frequency = 0.5, Amplitude = 15.0

2.3 Octaves

Lastly, we add noise octaves, a method that will add depth to our terrain. This is done by creating and adding multiple layers of noise with different frequencies and amplitudes. Starting off with a base frequency and amplitude, for each layer these two factors will be increased or decreased by a fixed amount and used when calling the noise function. In the end, the sum of the noise function calls will be the final value stored in the height [9].

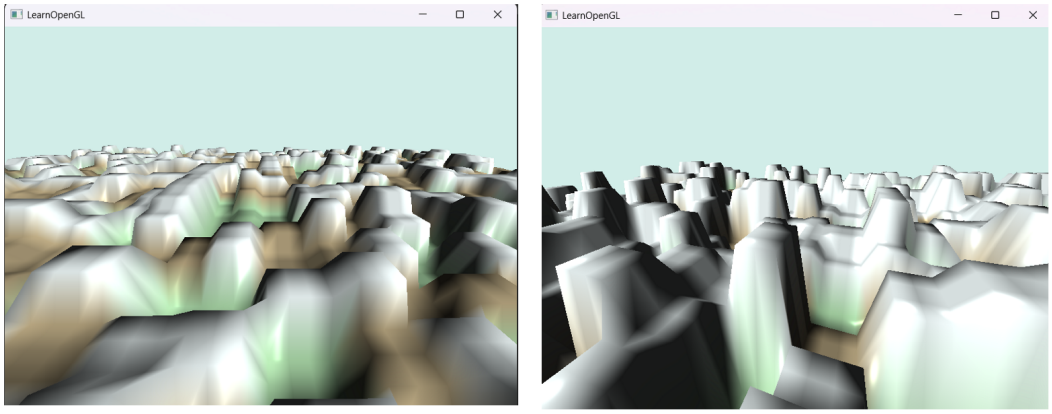

Figure 6: a) 1 octave, b) 6 octaves

3 Future Work

The project can certainly be extended in several different ways. Most interesting would be to implement other types of noise functions such as the aforementioned Perlin noise in the introduction, one of the most common algorithms within terrain generation. The Perlin noise algorithm differs from the current noise function in that it instead assigns unit vectors pointing at random directions for the four corners, and thus the interpolation step is also different as it handles vectors instead [6]. The advantage of Perlin noise over the current simple implementation is that it produces even smoother and continuous slopes. As seen in the above images, the slopes can still be quite blocky.

The choice of smoothstep function can also be played around with, as there are numerous to choose from. It is probable that there is one that could help provide even smoother slopes even with the current noise function, in which case we need to modify only one singular line of code and see the changes it brings. Different smoothstep functions are listed by Quilez in their article about the subject [7].

Finally, away from the subject of noise and terrain generation, the lighting in this specific project is not entirely correct, due to how normals are generated. Since this was not the main purpose of the project (terrain generation), this bug couldn't be fixed in time. The solution is most likely to compute the normals based on the height map. Additionally, to make a proper function for terrain coloring, use gradients and interpolate between the colors.

References

[1] Kenny. *Procedural Generation: An Overview*. Medium, Feb. 2021. URL: https://kentpawson123.medium.com/procedural-generation-an-overview-1b054a0f8d41.

[2] *Heightmap - Valve Developer Community*. Valvesoftware.com, 2022. URL: https://developer.valvesoftware.com/wiki/Heightmap.

[3] Muhammad Mujtaba. *Noise Functions*. My Explorations with Game Development, Dec. 2023. URL: https://gameidea.org/2023/12/16/noise-functions/.

[4] Newbie Indie Game Dev. *The Math Behind the Best-Selling Games: Perlin Noise*. YouTube, May 2025. URL: https://www.youtube.com/watch?v=MMj3WU4gORI.

[5] Jean-Colas Prunier. *Value Noise and Procedural Patterns*. Scratchapixel.com, 2016. URL: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/creating-simple-1D-noise.html.

[6] Zipped. *C++: Perlin Noise Tutorial*. www.youtube.com, July 2023. URL: https://www.youtube.com/watch?v=kCIaHqb60Cw.

[7] Inigo Quilez. *Inigo Quilez*. Iquilezles.org, 2025. URL: https://iquilezles.org/articles/smoothsteps/.

[8] Amit Patel. *Noise Functions and Map Generation*. Redblobgames.com, 2018. URL: https://www.redblobgames.com/articles/noise/introduction.html#frequency.

[9] Amit Patel. *Making maps with noise functions*. Redblobgames.com, 2016. URL: https://www.redblobgames.com/maps/terrain-from-noise/.