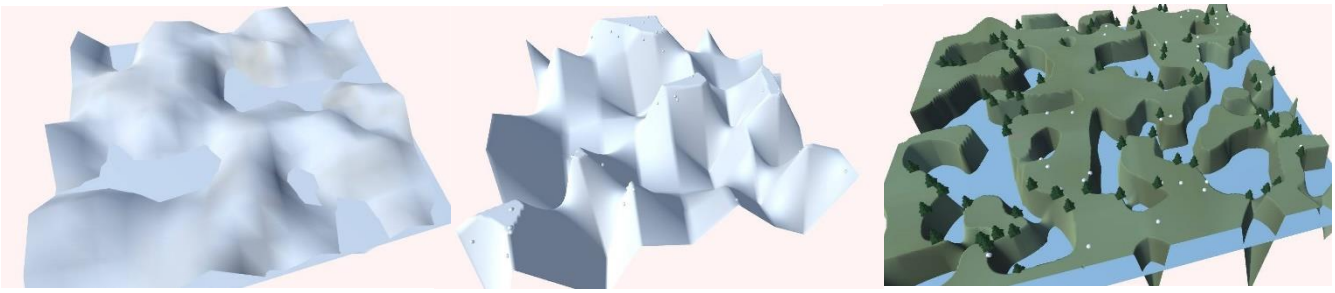
User-Based Procedural Generation of Terrain using Noise

Fanny Nilsson

March 7th, 2018



(a) ***(b)*** ***(c)***

Fig 1. Shows generated terrains at different times during the project. (a) Shows a terrain with high persistence value (smooth hills) and water added to the scene. (b) Shows a terrain with a high octave and a medium amplitude (fairly high hills). (c) Shows a terrain generated at the end of the project, with trees, water and snow. A high octave and a very high persistence value gives the map the opposite of hills, and instead large holes.

Abstract **- Procedural generation becomes more and more popular today and is used frequently in the game industry. The method is used for a large amount of areas within the industry but one of the most common ones are the building of terrain. Although procedural generation is a common method to generate terrain, it requires a programmer which can tie up a programmer's time in a team for quite a while. This paper will explain how to create a program that will randomly and procedurally generate a terrain map based on the user's input and constraints. This way not only programmers will be able to create the terrain maps fast, but the artists as well, which will lead to more freedom within the work force. Using noise algorithms such as Perlin Noise, and the user's input, every map will become different.**

Keywords: Procedural generation, terrain, user input, noise, height maps, fractal Brownian motion

I. INTRODUCTION

Creating games which rely on big maps and lots of freedom for the player to roam free is quite difficult since the replay value depends a lot on how the terrain and maps are built. To attract the players to continue explore, maps must be interesting and continue to surprise the players. They must give the player the sense of changing and offering new areas to explore. Achieving this is very time-consuming for artists to manually create each hill, tree, lake and mountain and therefore very expensive for the company. This is why procedural generation of terrain is becoming more and more popular. There are however a few problems with procedural generation as Freiknecht and Effelsberg describes in their survey paper (Freiknecht & Effelsberg 2017). Since a programmer – or a few programmers – must develop the algorithm, the terrain will not get the same believable look or feeling that the artists want, or the look it would get if artists themselves made it. This means that both artists and programmers must work on the terrain to just create a look that feels appropriate for each particular game or project. An approach like this is hardly the most effective and can cost the project quite a lot of time. This paper will explain how an application that uses noise and lets a user give input to create a pseudo-random terrain that fits the project, can be operated by either artists or programmers. The application developed in Unity uses a Fractal Brownian Motion algorithm to generate a noise which is used for a height map. The data from the height map is then applied to a terrain object to manipulate its terrain-data. By letting the user change variables such as the amplitude, octaves and persistence the application can give very different outputs. The combination of the randomness of the noise algorithm and the

input from the user creates a method that not only is very time efficient but also generates a high-quality product.

II. BACKGROUND

In this section noise will be explained and how it can be used to create random and believable terrain. Procedural generation will also shortly be explained.

A. Procedural Content Generation

A simple way of explaining procedural content generation (from here on called PCG) is that it is a way of "randomly" creating objects – for example terrain or textures – via an algorithm instead of creating it manually. The advantages are many but one of the biggest is of course how much time it saves as well as making a game much less predictable. Since complete randomness is not realistic, the methods that are used for PCG need to be pseudo-random rather than completely random.

B. Noise

Noise can mean a lot of things and for many people it is an unnecessary thing to simply ignore. However, in computer science, noise is very useful in many areas. One of those areas is creating terrain. There are many methods and algorithms for generating terrain with noise, but all these algorithms are pseudo-random which is needed in PCG to give the generated objects a feel of realism. It is very common for noise algorithms to be fractal but not all are. Those who are not natively fractal use a function called Fractal Brownian Motion (from here on called fBm) to simulate this. fBm is a method which adds several noises with different octaves – and therefore different frequencies and amplitudes – to create a multi-layered noise which increases the detail of the height map that is to be generated with the noise. The algorithm used in this paper is such an algorithm.

III. IMPLEMENTATION

Noise is basically just a bunch of random numbers and that's how the algorithm used in this paper started. An 2D array the size of the terrain was created and filled with a bunch of random numbers. Since complete randomness was not

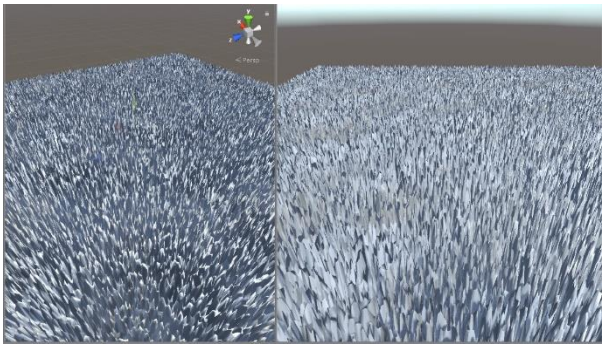


Fig 2. Terrain generated with completely random noise. Generated in the very beginning of the project.

desired this had to go through a stage of smoothening. Before the smoothing begins, as many noise arrays as the octave variable decides, are created. These will then begin the process of smoothing. By "zooming in" on the noise map it is made sure that all numbers will fit and work in this algorithm. Zooming in means that all numbers are transformed so they fit the algorithm. The octave variable decides how much the noise will be zoomed and it is shown by every, for example 5^{th}, 8^{th} or 12^{th} number on that index in the array becomes a new number. This is later "wrapped" so that when the end of one row (the x-axis) is reached it will continue from the beginning of the row but on the next column. During this time the interpolation variable, which will be used at the end to blend, is calculated as well. The code for the x-values and y-values look the same but the x-values are displayed here:

(Note: The integer octaveCounter is the input of the user and is a number between 0-9.)

```
int waveLength = (int)Mathf.Pow(2, octaveCounter);
float frequency = (1.0f / waveLength);

int x0 = (x / waveLength) * waveLength;
int x1 = (x0 + waveLength) % noiseWidth;

float horAlpha = (x - x0) * frequency;
```

Next the top corners are linearly interpolated with the horizontal alpha variable *horAlpha*. The variable *frequency* decides how frequent the hills will be. Same thing is done with the bottom corners. When this is done the two interpolated values that is the output is then interpolated with

the vertical alpha variable *vertAlpha* and saved in the array with now smooth noise, *smoothNoise*:

```
float top = Lerp(baseNoise[x0, y0], baseNoise[x1, y0], horAlpha);
float bottom = Lerp(baseNoise[x0, y1], baseNoise[x1, y1], horAlpha);

smoothNoise[x, y] = Lerp(top, bottom, vertAlpha);
```

Finally, each octave's smooth noise is added together but with a decreasing alpha value that is dependent to the variable *persistence*. This decides how much of each noise will be used and visible in the final result. The amplitude, which controls the heights the terrain can reach by using the noise, is multiplied with the persistence which decides how smooth or rough the transition between the noises will become. All amplitudes are then added and multiplied with each index of each noise array to create the final noise. This stage looks like this:

```
finalNoise[w, h] = smoothNoises[i][w, h] * amplitude;
```

The noise is then connected to a Unity Terrain object and manipulates the TerrainData to create a visual output. When the main part of the code is done, the user controls and constraints are added. These are very simple and consists of controlling the amplitude, persistence, octaves and a few terrain elements to make the terrain more realistic. The terrain elements are trees, snow and water. The user can here change the intervals of where each element is allowed to exist. This is solved by using Unity's UI elements such as sliders, buttons and input fields. The user can set constraints but only to a certain degree, since given to much freedom the application would stray a bit away from its original purpose as a terrain generator.

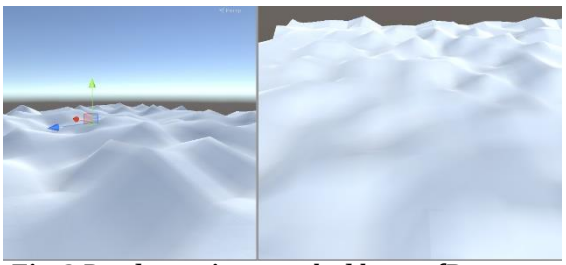


Fig. 3 Random noise smoothed but no fBm.

IV. RESULTS

What this project ended up as is a very simple but useful terrain generator which works in real-time and is easy for almost anyone to use. It produces a graphically plain terrain but simulates a fairly realistic one as well. The final implementation consists of almost all the things that were planned, and the visual output became even better-looking than thought.

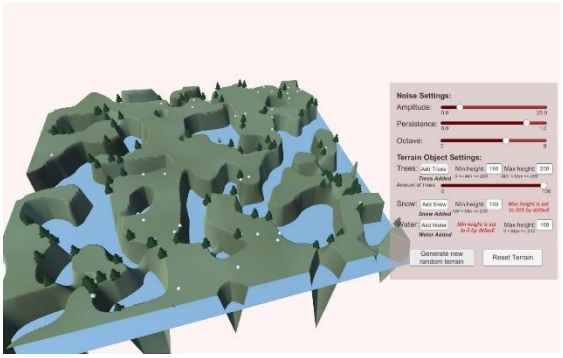


Fig. 4 Final result of application, with random terrain and UI for user controls.

V. DISCUSSION AND FUTURE WORK

Although the final result contains almost all the things that were planned the application was not built up quite the way that was expected. The plan was to first of all use the Perlin Noise algorithm to produce the noise. This however was shown to be a bit more complicated to implement (and understand) which therefore changed the plans a bit. The noise algorithm that is used is instead a simpler version of Perlin Noise, but not actual Perlin Noise. Secondly the plan was to – if time was given - implement a few other different noise algorithms to compare with the Perlin Noise. This did not happen and is therefore left for future work with this project.

VI. ACKNOWLEDGEMENTS

I would like to thank Robert Ringholm for the support and time spent brainstorming with me. A big thank you for the models of the trees as well.

REFERENCES

[1] Archer, T. (2011). *Procedurally Generating Terrain*. 1st ed. [pdf] Sioux City: Morningside College. Available at http://micsymposium.org/mics_2011_proceeding

s/mics2011_submission_30.pdf [Accessed 15 Jan. 2018].

[2] Parberry, I. (2014). *Designer Worlds: Procedural Generation of Infinite Terrain from Real-World Elevation Data*. 1st ed. [pdf] Dallas: University of North Texas. Available at https://pdfs.semanticscholar.org/9d3a/526cb09b9f7cd4cd9ecad835011899a1f907.pdf [Accessed 15 Jan. 2018].

[3] Freiknecht, J & Effelsberg, W 2017, "A Survey on the Procedural Generation of Virtual Worlds", *Multimodal Technologies and Interaction,* vol. 1, no. 27 Available at: http://www.mdpi.com/2414-4088/1/4/27 [Accessed. 5 March 2018].

[4] https://www.redblobgames.com/maps/terrain-from-noise/ (24/1-1/3 2018)

[5] https://www.redblobgames.com/articles/noise/introduction.html (24/1-1/3 2018)

[6] http://devmag.org.za/2009/04/25/perlin-noise/ (24/1-1/3 2018)