Procedural 2D Island generator

Jáchym Křeček
ČVUT–FIT
krecejac@fit.cvut.cz

December 30, 2022

This report was made for the semestral project of the BI-PYT course of B221

1 Introduction

Everywhere around the world we can witness different forms of nature like clouds, patterned textures like marble or shores of islands. All of these above can be remade with the use of fractals. One of these methods of using fractals for generating the landscape is for example the use of noise maps.

So the main principle of the work was creating an algorithm for generating 2D bitmap island maps with the use of such noises. There is a huge variety of different approaches with generating random terrain. We can get inspired by works in gaming industry such as Minecraft or Terraria. Minecraft for example uses fractal noises, white noises and many others.[3] Basically any game with an open world has some sort of randomly generating terrain with the use of noise. I decided to use the famous Perlin noise created by Ken Perlin in 1983.

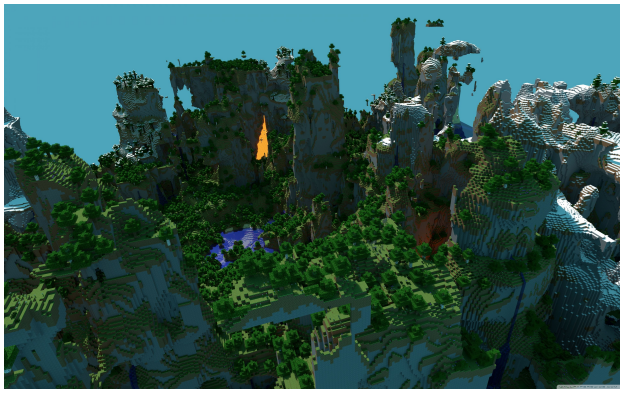

Figure 1: We all played Minecraft at some point

I made a program that takes input from the user in the form of adjustable switches, sliders and entry boxes and proceeds with the algorithm of Perlin noise. After that it displays the PNG image. Apart from that the program can display interactive 3D plots with height and moisture noise maps.

2 The Algorithm of Perlin Noise

In this section I will shortly describe the basic idea behind the algorithm. I added the whole explanation in the references. [2]

2.1 Grid Definition

An implementation typically involves three steps: defining a grid of random gradient vectors, computing the dot product between the gradient vectors and their offsets, and interpolation between these values.

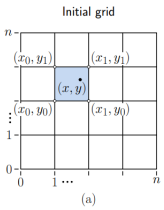

Figure 2: Grid definition [2]

Define an n-dimensional grid where each grid intersection has associated with it a fixed random n-dimensional unit-length gradient vector. [2]

2.2 Dot product and Interpolation

Now we have unique grid cells. We identify the 4 corners of the cell and their gradient vectors which is vector connecting the x,y coordinates and the corner. We then calculate an offset vector which is a displacement vector.

The first vector is the one pointing from the grid point (the corners) to the input point. The other vector is a constant vector assigned to each grid point. For each corner, we take the dot product between its gradient vector and the offset vector to the candidate point. [1]

Then we interpolate between those 4 dot products and we have a final result. The result will be in the range [-1.0, 1.0]. [1]

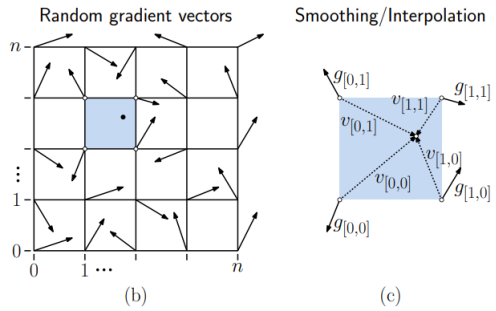

Figure 3: Interpolation and gradient vectors [2]

3 Elevation and Moisture

Now that we have the boring math out of the way we can focus on the exciting part of the program. This part was more about tweaking small numbers and took up the most of the work. Finding the right value to fit for my interval of values was a tedious but fun task. The basic idea is that we take these generated values (always based on the seed) and assign them some biome. There are two noise maps. Elevation and moisture. These two mix and create the final biome.

There are about seven different modes each favors different biomes. For example the planet of Dagobah will have much more jungles than for example Terra. Or Tatooine will have more desserts etc.

The whole grid was saved as 2D numpy and in the assign function I translated the values into a RGB tuple. The result of this was passed into the plotting and image viewing functions.

4 Plotting and Image viewing

And the last part of the app was displaying such values in a graph or image. I used PIL for bitmap and mathplotlib and plotly for 3D surface plots. There is also a feature for saving each image and figure created.

One of the problems I encountered was the very weird way of plotting 3D surfaces in mathplotlib when we have only a 2D numpy array. This was eventually solved with creating two linspaces of x and y of rows and collumns of the matrix and making a meshgrid. The x, y and the whole numpy array was then passed to the plotting. Other than that this was quite OK.

5 Optimization and speed

The number one issue of the whole project was the speed. The library of noise I used had a pretty nice implementation but when I tried using my own, the

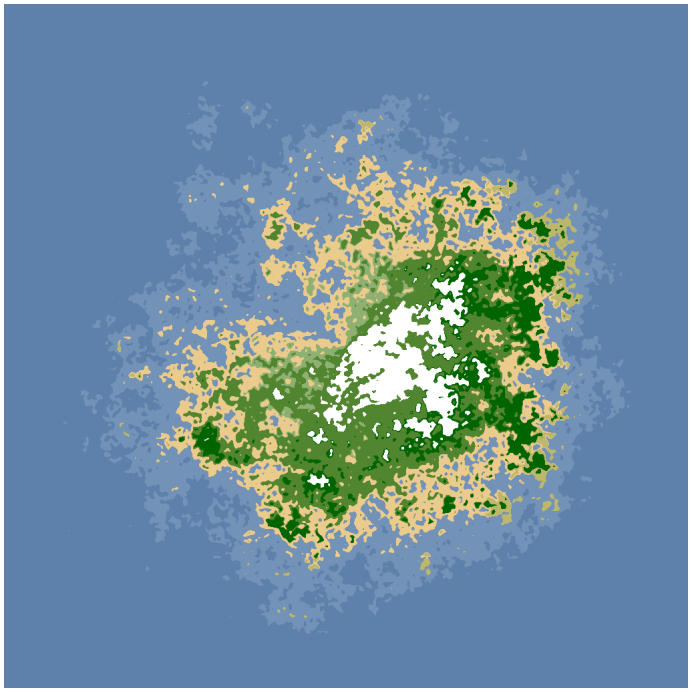

Figure 4: Result of the algorithm with high persistance

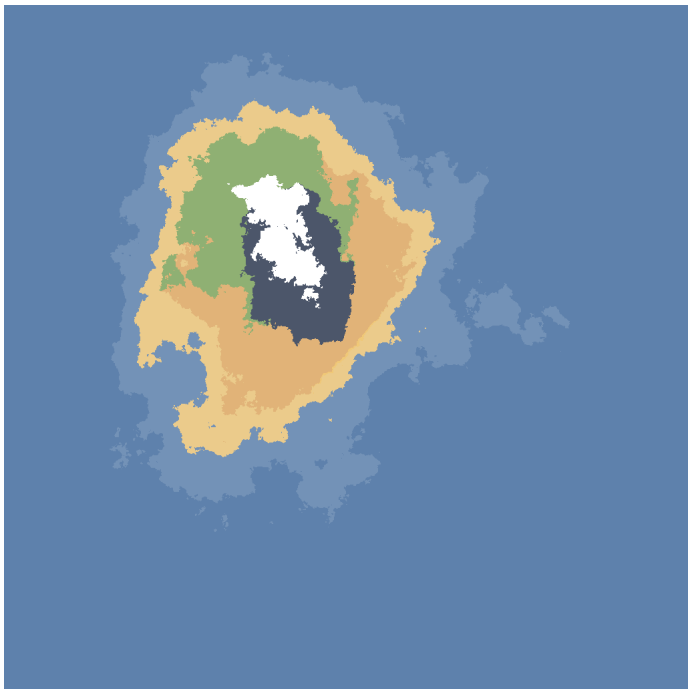

Figure 5: Lower persistance but more octaves

results were much slower. On higher octaves it's pretty problematic. I tried to use more list comprehensions in the functions and even got inspired by the reference implementation of Ken Perlin algorithm for my custom function but to no avail. This is something I will work on and hopefully manage to improve.

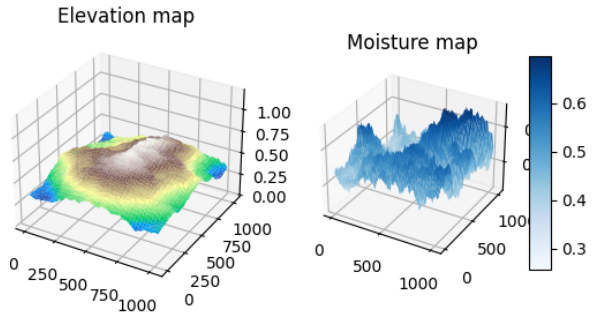

Figure 6: The 3D plotting. Moisture and Height map

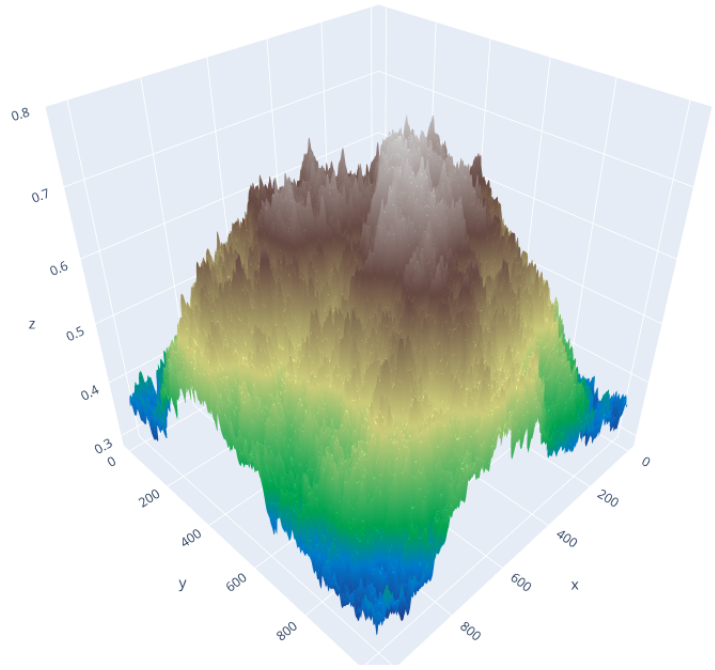

Figure 7: The mapping of surface. Plotly

6 Conclusion

This whole project was very fun. I am fascinated by fractals and this work was a great way of exploring this exciting topic! I give my thanks for the mentors/teachers in my school that gave me the idea and possibility to implement this in a real code. This is something I will definitely work on in the future. Maybe adding rivers and trees to this map would a be a great addition. Or moving towards 3D noise. The possibilities are many. Other than that I am very pleased with the results

References

[1] Perlin Noise . Perlin noise — Wikipedia, the free encyclopedia. online, 2022. [cit. 2022-30-12] `https://en.wikipedia.org/wiki/Perlin_noise`.

[2] Dave Mount. Cmsc 425: Lecture 14 procedural generation: Perlin noise. online, 2018. [cit. 2022-30-12] `https://www.cs.umd.edu/class/fall2018/cmsc425/Lects/lect14-perlin.pdf`.

[3] Alan Zucconi. The world generation of minecraft. online, 2022. [cit. 2022-30-12] `https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/`.