1 Overview

This report presents our advancement on the first part of the project : terrain generation using procedural methods. Figure 1 shows an example of what our actual code base is able to generate. All the minimal steps to display a procedurally generated terrain were successfully completed. We did also implement some other optional suggested methods, like the multifractal and the simple noise. We still plan to implement more.

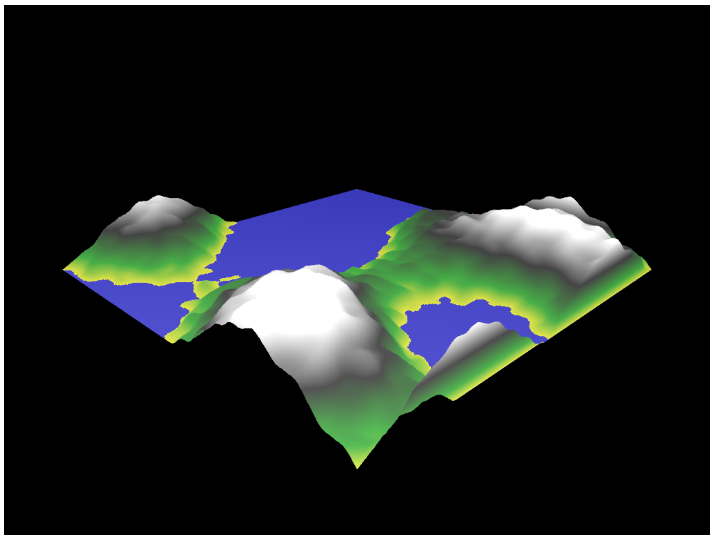


Figure 1: Example terrain generated by fractal Brownian motion with $H = 1.1$, $l = 10$ and 10 octaves, Perlin basis noise function.

The main difficulties we did encounter during this first stage were about the framework. As for practicals and homeworks everything was set, we did not have any experience on the various functions used to create, bind or copy data into the various OpenGL object types. We took a lot of time to figure out how all this was working together. Now that we acquired some experience, the implementation of the next steps will surely be facilitated.

We did also loose time with some oddities. Let us give an example. The Perlin noise permutation table contains values from 0 to 255. It would thus make sense to store them as a `GL_ubyte` array and to pass them to the shaders as a 1D texture of internal type `GL_R8UI`. No data were however passed. The glsl `textureSize` built-in function was always returning a size of 0. After hours of trying to understand why a simple buffer copy does not work, we found a little note which said that standard compliant implementations do not have to exactly match the provided list of internal types and can fall back to other types[1]. After trying some other types like `GL_R32I` we resigned and used `GL_R32F` at the expense of 2 bytes per texel. The exact cause of this problem was however not elucidated.

As noted in the hand-out, we add to the fact that OpenGL debugging is hard and time-consuming. We use a lot of "black box" functions and debugging is not very practical. This is especially true for glsl and the infrastructure for data passing, as there is nothing until it works. At least at the beginning. When the infrastructure is in place and an output buffer is available, "printf debugging" can be used.

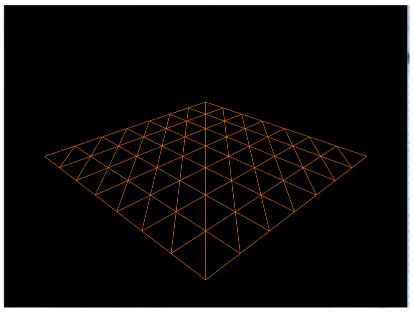
[1]Explained on the OpenGL wiki.

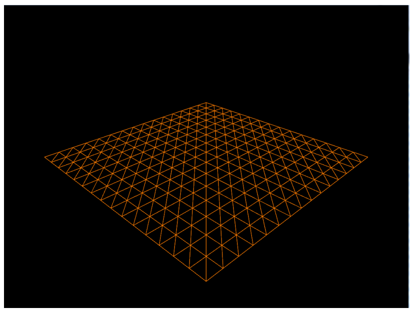
2 Implementation

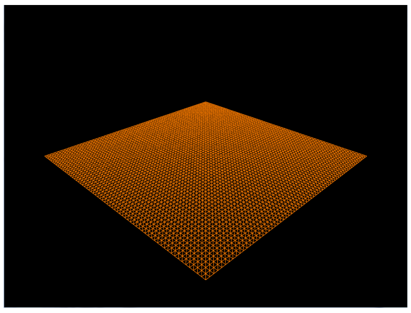
2.1 Triangle grid

We first generate the flat triangle mesh as a base for our terrain. The implementation is quite straightforward for a grid of size $N \times N$ as first, two VBOs storing the positions of each vertex and the corresponding faces' indices also are calculated.

We then render it as a triangle mesh by function `glDrawElements` to get a smooth triangle mesh (fig. 2d). It is also worth noticing the use of `glPolygonMode`, which allows us to render only the triangles' boundaries as shown in figures 2a, 2b and 2c for $N = 8$, $N = 16$ and $N = 64$, respectively.


(a) Lines with $N = 8$


(b) Lines with $N = 16$


(c) Lines with $N = 64$

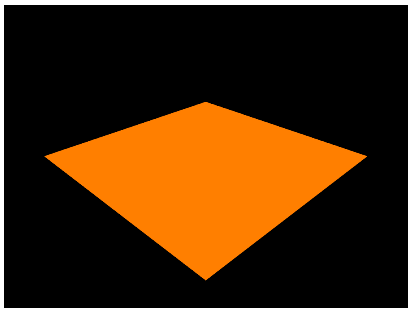

(d) Triangles

Figure 2: Triangle grid

2.2 Diffuse shading

The terrain generated from displacing each vertex in base triangle grid according to height map will be applied the diffuse shading for better visualization. In short, each vertex will be colored by the result value of following equation: $I_d k_d (N \cdot L)$ where I_d and k_d is the diffuse color of the light source and material, respectively. N is the normal vector at that vertex and L is the normalized light direction vector.

We define te diffuse color of light source and material as well as the light position so except N, other values are very easy to compute. For calculating normal vector N, we use finite difference to approximate a gradient vectors ∇x and ∇y along x and y directions. Note that to compute theses gradients, we need the elevation at four neighboring vertices as well, which can be done again by looking up on the height map that we generated.

Then, the normal vector will be cross product of ∇x and ∇y.

2.3 Heightmap texture

We did first construct an heightmap by hand (`gen_test_height_map`) with a 3x3 pixels texture to test the displacement procedure of the rendering vertex shader and the height sensible coloring of the rendering fragment shader (fig. 3a). We then implemented the texture rendering (`gen_height_map`). To test the framework, we rendered a simple texture of 1024x1024 pixels which all have the value of 0.5 (fig. 3b).

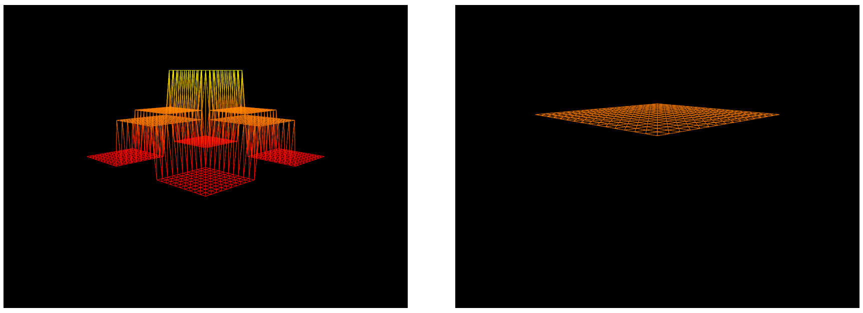

(a) Heightmap constructed by hand (b) Simple generated heightmap

Figure 3: Heightmaps

2.4 Perlin noise

We then implemented the Perlin noise in the heightmap fragment shader. As suggested by the tutorial[2], we used a permutation table to generate a pseudo-random number per square. The Fisher-Yates shuffle[3], also known as the Knuth shuffle is unbiased, as long as the underlying random generator is.

We use the `texelFetch` function to access the texture by its index instead of the normalized coordinate.

2.5 Fractal Brownian movement

Figure 4 shows the variety of procedural terrains that can be generated by an unique method, only by altering the seed of the pseudo-random number generator.

2.6 Turbulence

We also try to implement turbulence, which is very similar to fBm but use absolute value of noise instead. The result is shown in figure 5.

[2]The GPU Gems 2 tutorial given in the hand-out
[3]Described on Wikipedia.

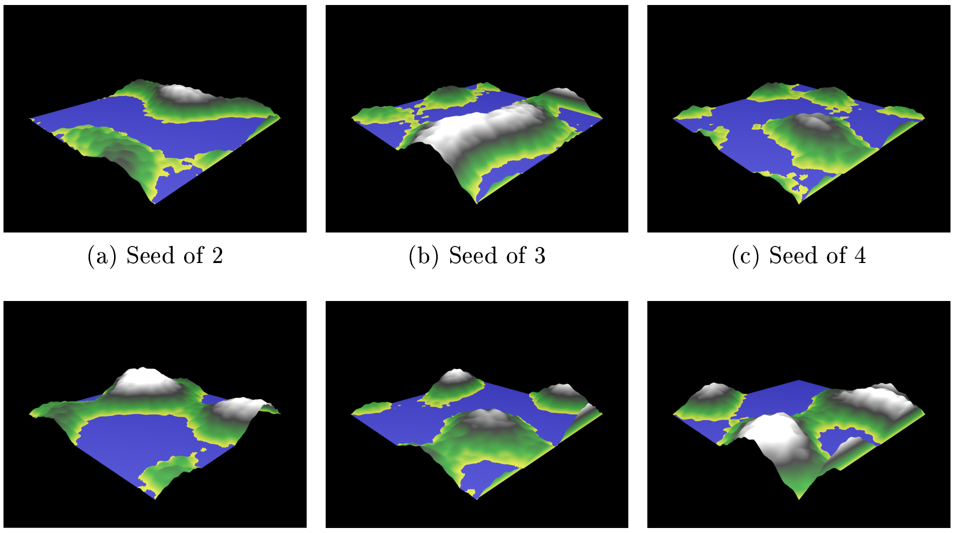

(a) Seed of 2 (b) Seed of 3 (c) Seed of 4
(d) Seed of 5 (e) Seed of 8 (f) Seed of 10

Figure 4: Example of terrains generated by fractal Brownian motion with $H = 1.1$, $l = 10$ and 10 octaves, Perlin basis noise function. Different seeds were used to initialize the permutation table random shuffle.

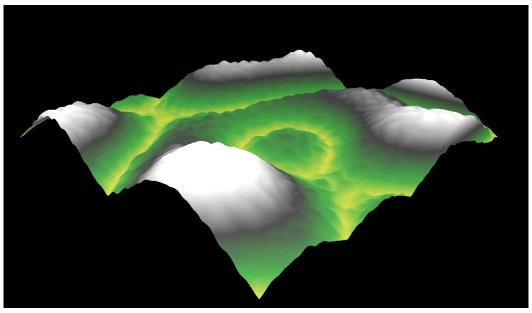


Figure 5: Example of terrain generated by the turbulence method.

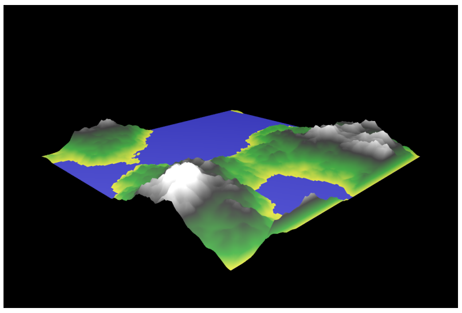
2.7 Advanced topic : Multifractal

As an enhancement to the fractal Brownian motion, we implemented the multifractal algorithm. to generate a terrain using Perlin basis noise function or Simplex basis noise function. Figures 6a and 6b show examples of such generated terrain using Perlin and Simplex basis noise functions.

2.8 Advanced topic : Simplex noise

For Perlin noise, we leverage on a square grid (4 corners), which is sometimes unnecessary. Simplex noise, in contrary bases on a **simplex** grid, which in our 2D case is an equilateral triangles (see figure 7a).

Since it would be very computationally expensive to find the position of three surrounding neighbors (colorfully encircled), we can skew the grid so that it can have the shape of square grid (Figure 7b). Once we skew the grid, it would be every easy to find the triangle we are in by simple test of x and y coordinate (figure 7c).


(a) Perlin noise basis

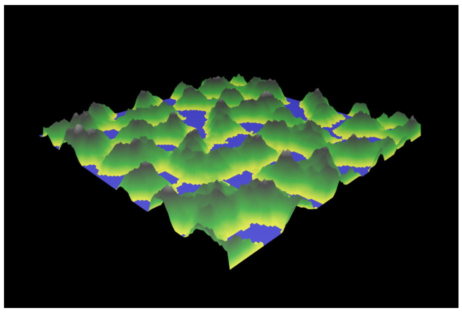
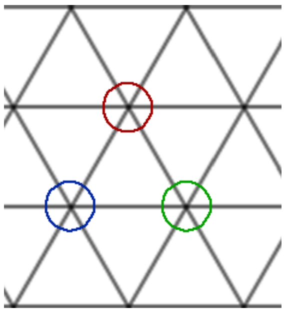
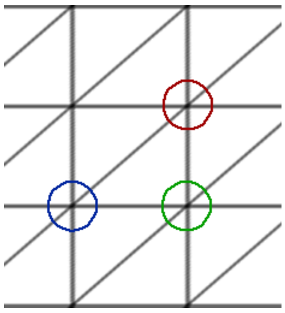
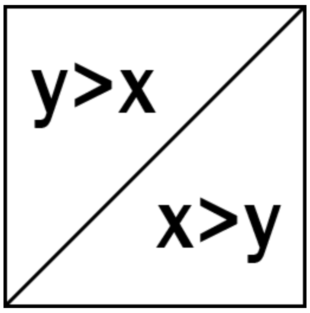

(b) Simplex noise basis

Figure 6: Example of terrains generated by the Multifractal method.


(a) Simplex grid


(b) Skewed simplex grid



(c) Determine current equilateral

Figure 7: The simplex grid and its skewed version used for constructing Simplex noise.

After retrieving the coordinates of three surrounding corners, we use the same pseudo-random process as used in Perlin noise to look up for gradient of each corner. Then we find the contribution of each corner, which is proportional to the cube of the distance $(0.5 - x_d^2 - y_d^2)$. Here x_d and y_d is the difference in x and y axis from the corner to the position we are considering. Also, the proportional ratio is the dot product of the random gradient and the position vector.

Finally, we just sum up the contribution of each corner and scale it by some scalar (in our case is **80**).

Figure 8 shows an example of a terrain generated by the implement simplex noise function.

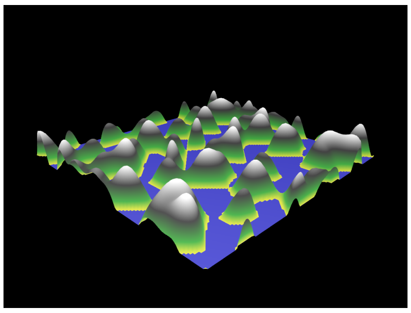


Figure 8: Example of terrain generated by simplex noise.

3 Results

Figure 9 shows a closer look at the rendered triangle mesh.

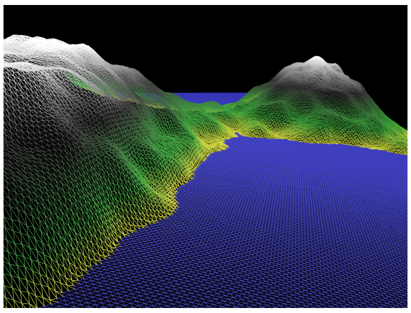


Figure 9: A closer look at the triangle mesh.