Stage 1 : Terrain Generation

Group4 : Dongge Han Chervine Majeri Nicolas Perdu

April 14, 2014

1 Overview

In this stage of the project, the goal was to generate a terrain, using procedural techniques, so that building different terrains would be a matter of adjusting a few parameters, instead of painstakingly doing everything manually.

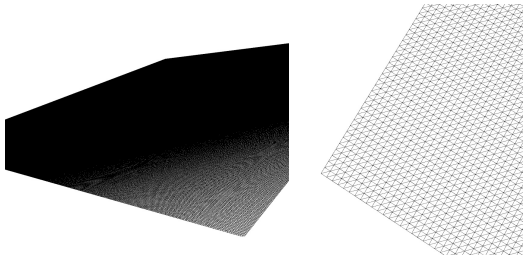
The idea was to generate a height-map, which would then be used to displace the z-coordinates of a plane.
To this end, we used a smooth noise function (Perlin, Worley), which, when combined with itself in various ways (fractional Brownian motion, weighted or not), and mapped onto a texture, gives us a good instance of a height-map.

Basic diffuse lighting was also implemented, using the finite difference method to compute the normals.

2 Implementation

2.1 The plane

The plane is made up of tiny squares. Which are made up of triangles. The implementation simply computes the vertices and puts them in a vector object.



(a) Far-away (b) Close-up

Figure 1: The plane

2.2 The texture

Now that we have our plane and an OpenGL context, we can work on generating the textures.
At first, a framebuffer wasn't used, so the results could be seen on screen.
We simply draw a square reaching from the bottom left to the top right, and we can map the noise function onto it with the fragment buffer.

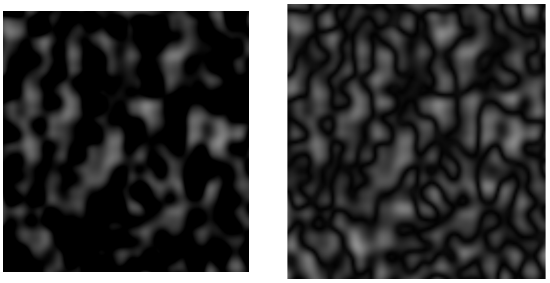
2.2.1 The noise

The goal is to implement the noise function.
Perlin noise works on a positive coordinate plane separated into small squares that the vertex will be in.
We need to compute a random gradient for each positive integer-indexed point in space.
However, once these gradients are set, they need to remain the same.
So true randomness is out of the question.
We need a pseudo-random number generator. and since glsl has no built in PRNG, we have to make our own.
A possible approach would be to generate 256 random gradients, and then use the hash of the (x, y) coordinates of the integer point to get the definitive gradient.
To hash the (x,y) coordinates, we can use a permutation table of [0, 255] and then compute (perm(x + perm(y))).

The final gradient for point(x,y) in space would be grad(perm(x + perm(y))).

Implementation-wise :
Suppose we are treating the vertex with position P =(x, y).
We need to transform this vertex into a vertex into a representation that Perlin noise can work with. So x $\in$ [0, nX] and y $\in$ [0, nY]. So we simply do P = (P/2 + 0.5)*vec2(nX, nY). And we can compute our noise at position P. As far as the gradient table and permutation tables go, they are sent to the fragment shader by putting them inside 1D textures, with internal format GLR32F, type GLFLOAT and format GLRED.

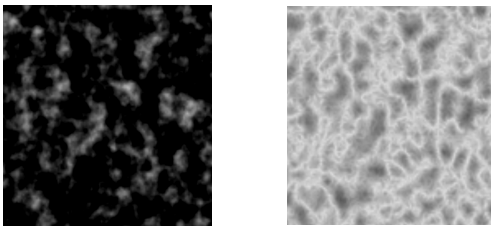
The vertex shader provides us with the vertex position.



(a) nX = nY = 25 (b) 1 - abs(Perlin(x,y))

Figure 2: Perlin noise

We also implement the fractional Brownian motion and its weighted form to generate the heightmap.



(a) H = 1.0, Lac = 2.0, fBm of perlin (b) H = 0.5, Lac = 3, Hmf of 1-abs(perlin)

Figure 3: Fractal functions

2.2.2 Framebuffer

After seeing what the texture looks like on the screen, we can feel confident drawing it on our Framebuffer.
To achieve this, we bind a new framebuffer, and a new texture noiseTexture which we bind onto the framebuffer's first drawing slot.
And done.

2.3 The terrain

Now we have a plain plane, and a height-map stored in a texture. We can simply put all of it together, and start creating terrains.

In our vertex shader, we compute the normal, and we displace the vertices, in our fragment shader, we colour according to height.
Here are some example terrains :

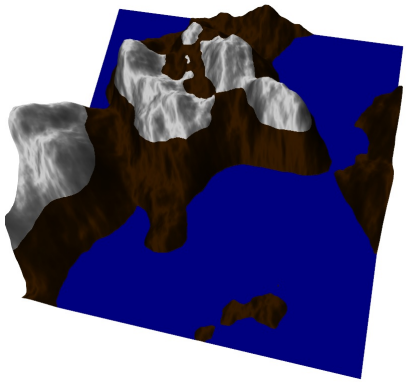


Figure 4: H = 1.0, Lac = 2, fBm

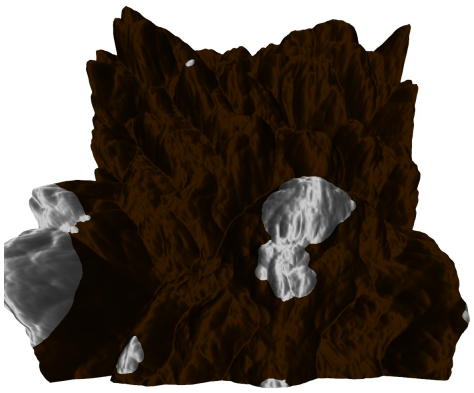


Figure 5: fBm, H = 1.0, lac = 2.0, seed = 466

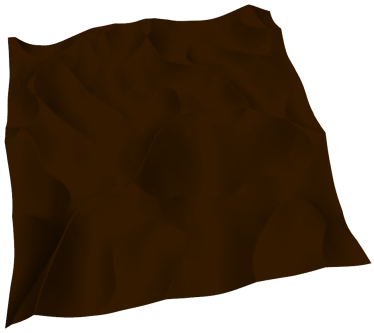


Figure 6: H = 0.5, Lac = 3, Hmf of 1-abs(perlin)

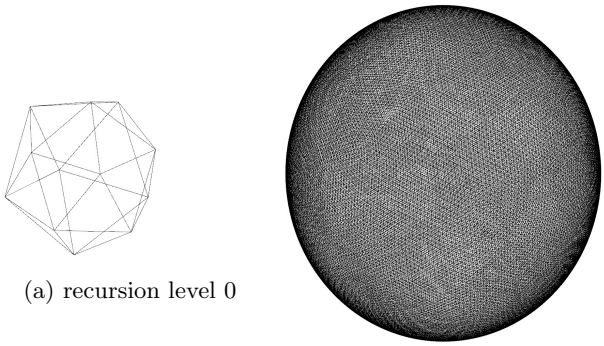
3 Advanced

3.1 Planets

The idea of generating planets is quite appealing, but mostly, it is a logical next step to the generation of the basic 2D terrain.

3.1.1 The sphere

We start out with an icosahedron, and we divide every triangle that's constituting it into 4 smaller triangles. We repeat until we have a nice sphere.



(a) recursion level 0

(b) recursion level 6

Figure 7: Spheres

3.1.2 The texture

The step of transforming our 2D texture into 3D textures is quite easy : indeed, Perlin noise in 2D is very similar to Perlin noise in 3D.
We just need to think of cubes instead of squares.
Furthermore, we simply need to draw as many squares as we want on different levels of textures to turn our 2D texture into a 3D one.

3.1.3 The terrain

Generating the terrain is a little trickier, however.
We need to displace the height of the point at P according to the normal on this point of the sphere, which turns out to be P.
So : P, becomes fP = P + height(P)*normalize(P).

Now for the lighting, we need to compute the normal.
The idea here is to compute the normal as if we're on a plane, then rotate this normal to its correct value.
First a few definitions :
θ = angle between z and P, it is between 0 and 90 degrees.
Knowing P, we can deduce : $cos(\theta) = Pz$, and then deduce $sin(\theta) = \sqrt{1 - cos(\theta)^2}$.
The normal on the very top of the sphere is the same as the normal on the plane (the normal of the sphere point z=1 is the same as the normal of the point if it were on the plane z = 1), we need to rotate it to where P is, which will give us the correct normal.
This operation is a rotation of angle θ around axis $P \times z$.
Such a rotation can be expressed by the following rotation matrix :
rot =

$$\begin{pmatrix} cos(\theta) + Px^2 * (1 - cos(\theta)) & Px * Py * (1 - cos(\theta)) & Py * sin(\theta) \\ Py * Px * (1 - cos(\theta)) & cos(\theta) + Py^2 * (1 - cos(\theta)) & -Px * sin(\theta) \\ -Py * sin(\theta) & Px * sin(\theta) & cos(\theta) \end{pmatrix}$$

After the rotation we have the correct normal of any point on the circle.

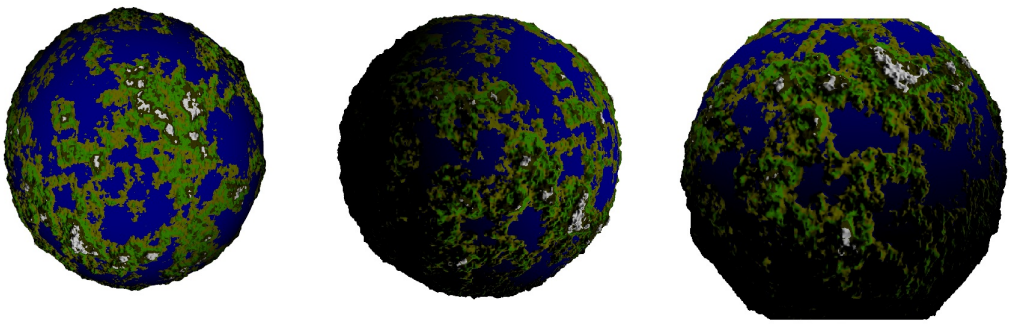


Figure 8: The same planet, from different angles. The light-source stays the same place relative to the planet. Obtained with fBm and perlin, nX = nY = nZ = 8, H = 1.0, Lac = 2

3.2 Worley noise

So Perlin noise isn't the only smooth noise function out there, there are many others. Worley's cell noise is one of them.
It works this way :
We have our space. We divide it into integer-indexed cubes (very similar to Perlin thus far).
In our cube are a random number of points we call "Feature Points" (We've set

our number of points to be 1 + Poisson distribution).
Those feature points are set as random, but again, they need to remain static throughout the application of the algorithm.
One way to ensure that they are is to use (again) a permutation table, coupled with an x, a y and a z coordinate-tables.
Now the only thing the algorithm does is store the distances to the n points closest to our point.
Here, we only need to look at most at the neighbouring cubes. Since the number of feature points per cube is at least 1.
This makes 27 cubes to look at and could be done more efficiently, but efficiency isn't the goal at this point.

Now we can get images using the distances we have. Here are a few.

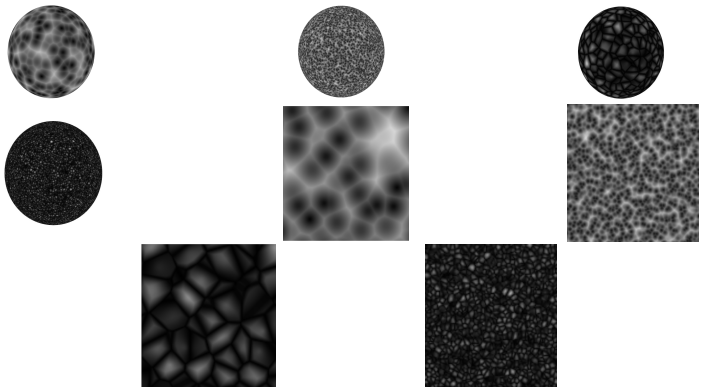


Figure 9: Worley cell noise.

4 Remaining to do

Ranked by interestingness :

A generated planet is a good thing, a generated planet which we can zoom in on, and still have nice resolution is a better thing : LOD-rendering with tesselation shaders will probably see the light of day, some day.

Trees, procedurally generated, swaying, luscious trees.

Waves : Perlin noise is used in particular to simulate wave movements, let's apply it ! (Not to be expected)

Unlimited terrain : Welcome to the world of memory management, where things need to be deallocated and stored in files !

5 Performance

Most of the operations are done in linear time. Memory management is not an issue since the vertices are allocated once, and then never not in use until the end of the program.
On my nVidia GTX680M, with textures of 256x256x256 and a recursion level of 7 (ie $60 * 4^7$ vertices), the render time is about 5 seconds.