1 Overview

This report presents our advancement on the second part of the project : rendering using texture mapping and shadowing.

2 Implementation

2.1 Basics

2.1.1 Texturing

In this part, our idea is that we can use whatever texture that we want, whatever method or way of blending that we want as long as the terrain looks realistic. Also, the way we implement texture blending is trial and errors, we try many different ways of blending and continue in the direction that generates better results.

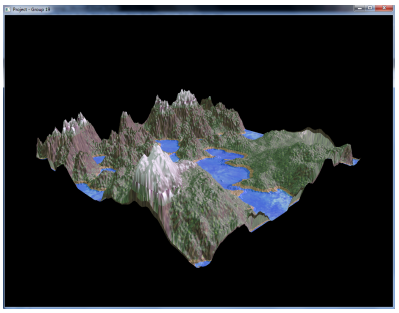


Figure 1: The terrain with blended texture.

2.1.2 Modeling the sky

For this part, the idea is very straightforward as we try to render two objects : terrain and a textured cube covered the terrain and display them together on the screen. However, it is very time-consuming to implement but good things are after finishing this part, we can gain a very solid understanding of OpenGL vertex buffer mechanism.

To render multiple objects together, before rendering each object, we set its frame buffer as a render target and bind the vertex array of that object. In addition, we use separate program to render each object, so before drawing it, we need to call `glUseProgram` in OpenGL.

To texturize the skybox with sky images, we make use of `GL_CUBE_MAP_SEAMLESS` option in OpenGL to get rid of the problem with seams. We also note that for high-resolution texture files, which are $1024 \cdot 1024 \cdot 3 = 3\text{MB}$ each, are too big to be allocated on the stack so they need to be placed also on the heap.

The results of this part is shown in Figure 2.

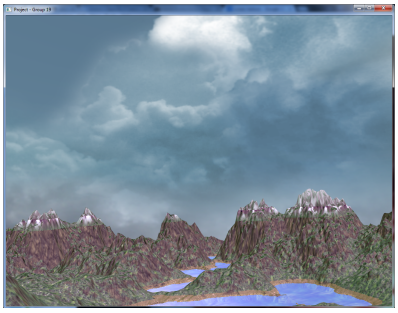
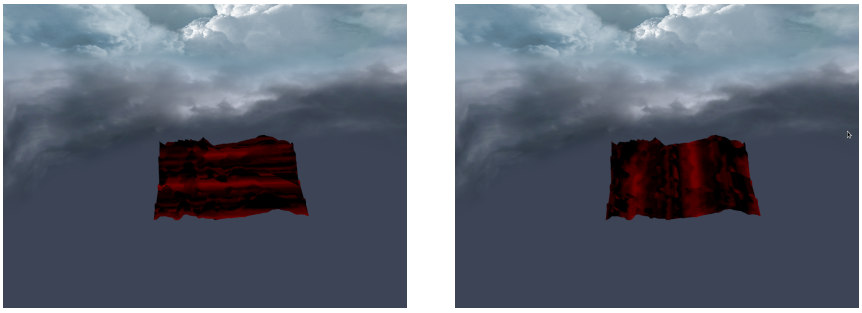


Figure 2: The terrain covered by a sky box.

2.1.3 Self shadowing

The shadow mapping is implemented by a third C++ class : the `ShadowMap`. See section 3.1 for a discussion on the object oriented refactoring of our code base. This class shares the same Vertex array object (thus the vertices and indices buffers) as the `Terrain` class. It renders to a depth texture which is then passed to the `Terrain` class as an input texture. Figure 3 shows this texture for two different light source positions.



(a) Light source at (0,3,0) (b) Light source at (3,0,0)

Figure 3: Shadow maps

2.2 Advanced

2.2.1 Approximating water reflections/refractions

2.2.2 Water depth effect (participating media)

2.2.3 Water dynamics

For this part, we follow a very simple approach. In the vertex shader of the terrain generation, for only vertices belonging to the water region, we displace it in z-direction by a sin function of x and y position (up to a scale). We also pass a uniform variable `time` to control the phase of the displacement. Even this is super simple but the result is pretty good.

In addition, we also do normal mapping for water region. Instead of calculating the normal like other regions, we do a normal vector lookup in the normal map and use that vector for shading.

2.2.4 Time of the day

3 Improvements on last stage

3.1 Code modularization using object oriented programming

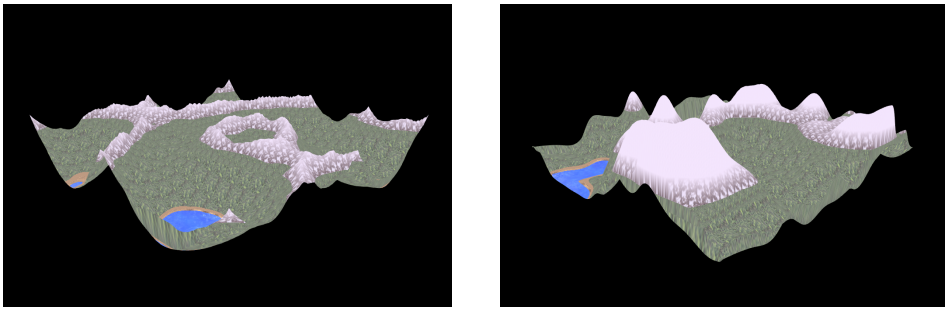
We re-factored our code base to use an object oriented paradigm. This allows better modularization and generalization. The idea is to have a base class (`RenderingContext`) who defines the interface and implements common methods. The contexts (`Terrain`, `Skybox` and `Shadow`) then inherits from it and implement further specializations. This simplifies the manipulation of the contexts from the main program as they all have the same interface. A common implementation also limits code duplication.

This is similar to what is proposed in the hand-out. With however a major difference : we wanted to separate the class declaration and definition (i.e. implementation) in a header and a source file. There is two motivations behind this : putting the declaration in a header makes it easy for the class user to identify methods and parameters he can use; putting the definition in a separate compilation unit isolates the code.

The problem we encountered with this approach is that the OpenGP headers (included from `common.h`) do not contain only declarations, but also definitions (i.e. code). Such that we cannot include the `common.h` header in more than one compilation unit. If we did so, the linker would complain appropriately about multiple definitions of the same functions. We thus created an `opengp.h` header who declares (and not defines) the functions and constants from OpenGP used in our project. Their definitions (included from `common.h`) are solely included in the main compilation unit. This is a workaround but it allows to use a proper object oriented paradigm with one declaration and one definition file per object without modifying the provided framework.

3.2 Parameter tuning for better terrain generation

In order to find a better terrain, several combination method has been tested. The first idea was to use an hybrid multi fractal method based on 1 minus the absolute value generated by perlin noise. After tuning parameters, we found a terrain counting few lacs and really sharp mountains as shown Figure 4a. Such terrain doesn't contain flat plain, therefor we tested a multi fractal method combining perlin and simplex noise as shown Figure 4b. Oppositely to the hybrid multi-fractal, this method generates flat plains but smooth mountains.

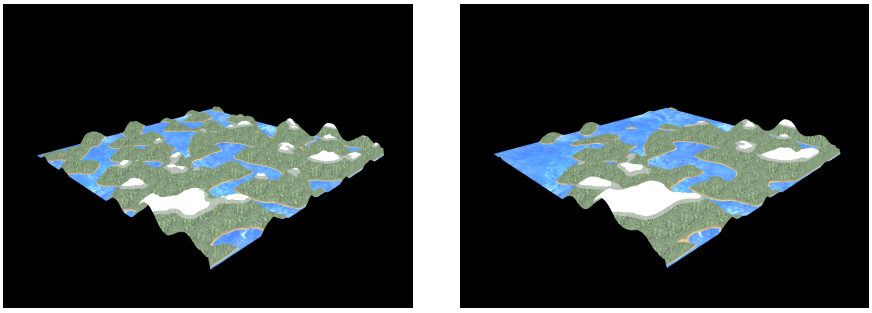


(a) Hybrid multi-fractal based on Perlin noise

(b) Multi-fractal based on perlin and simplex noise

Figure 4: Terrain generated

As we developed several generating method in project part 1, an idea is to combine them. Additive combination: sum the product of a coefficient and the height generated by fractal Brownian motion, multi-fractal (simplex noise based), multi-fractal(perlin noise based), simplex noise and perlin noise. The sum of coefficients must be equal to 1 in order to have a balanced and well scaled terrain. As we have 5 coefficients variating from 0.1 to 1.0, we have 102 possible combinations (for a sum of coefficient equal to 1). We used an external software called "autoit" to generate the 102 combinations. Using autoit, we wrote a loop setting the coefficients, generating the terrain and saving a snapshot of it. Two example of generated terrain using this method are shown Figures 5a, 5a.



(a) Example 1

(b) Example 2

Figure 5: Additive combination

Random combination: use different noise and multi-fractal methods according to a randomly generated number. Using randomly a different noise per pixel leads to a highly discontinuous terrain with abrupt change of height as shown Figure 6

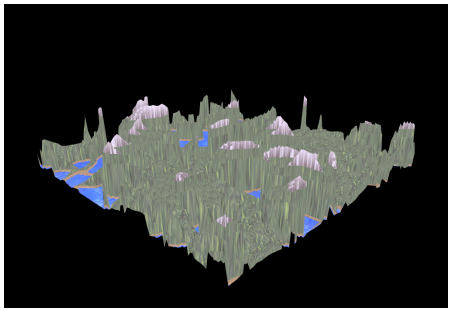
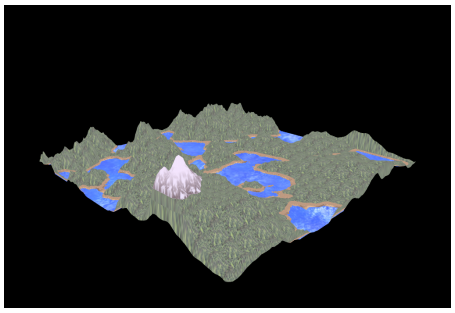
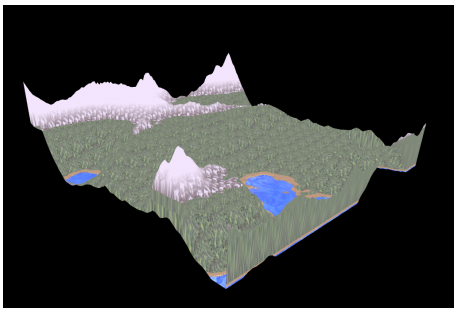


Figure 6: Random combination

Spatial combination: use several noise and multi-fractal methods multiplied by a coefficient computed according to the x or y position. This method will smoothly superpose different terrain generation methods creating sharp mountains with lacs and flat plains as shown figures 7a and 7b.



(a) Example 1



(b) Example 2

Figure 7: Spatial combination

3.3 Fixing normal vector calculation

The normal vector calculation implemented in previous stage is actually not correct but only until we try to blend the texture based on the normal vector that we recognize that the results are not correct. While it costs us a lot of time to re-implement normal vector calculation, the experience that we learned when correcting the normal vector is really invaluable for further understanding OpenGL pipeline.

Firstly, we map the world coordinate to a texel before using `textureOffset` function to look up the height of surrounding texels in both x and y direction. A finite difference is used to approximate the tangent vector to the surface at that point on the height map then finally, a normal vector will be just the cross product of tangent vectors along x and y directions.

In addition, the above normal vector is just in height map coordinate (ranging from $[0, 1] \times [0, 1]$ in xy plane) so we need to map it to our world coordinate of the grid ([-1,1] in xy plane). Note that when transform the normal vector, we actually need to multiply it with the inverse of transpose matrix of the transformed one.

To test the normal vector, we output it as the color of each fragment as shown in Figure 8. We can see that the result is quite reasonable, for example in the ground part as it is flat

region, the normal vector should be (0,0,1) and as a result, its color is blue.

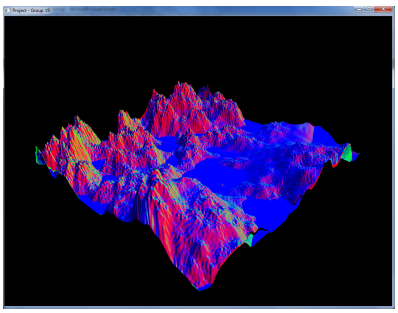


Figure 8: The terrain is colored by its own normal vector.

4 Results

In order to quantify the computation power needed and estimate the efficiency of our program, we proceeded FPS measurements. We measured the amount of frame per second in different case:

- Only terrain display with $N = 32$ (1024 vertices): 254 FPS
- Only terrain display with $N = 64$ (4096 vertices): 176 FPS
- Only terrain display with $N = 128$ (16384 vertices): 114 FPS
- Terrain and skybox ($N = 128$) : 108 FPS
- Terrain, skybox and water animation ($N = 128$) : 109FPS

We can observe that as expected the amount of frames per second decrease with the increasing amount of vertices. The skybox have a really low "cost" (decrease of 6 FPS corresponding to 5.2 %) and the water animation (just mooving the texture) is negligible.

5 Project managing tools

As we never work together in the same place, we need some efficient communication tools. During the first part of the project, we communicated per Email and reached over 100 mails. In order to have a clear communication, avoid time loss and improve clearness of our project progress, we used a tool called Trello (https://trello.com). It's a simple tool allowing list managing with a simple and efficient interface (figure 9).

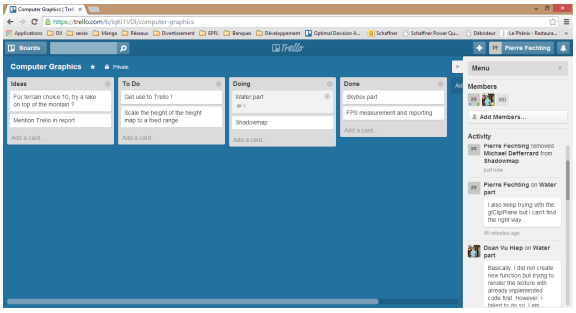


Figure 9: Trello interface