Procedurally Generated Planet Using a Multi-Layered Noise Function

Alexandre Marques Dias
University of Montreal
Montreal, Canada
alexandre.marques.dias@umontreal.ca

Pierre Poulin
University of Montreal
Montreal, Canada
pierre.poulin@umontreal.ca

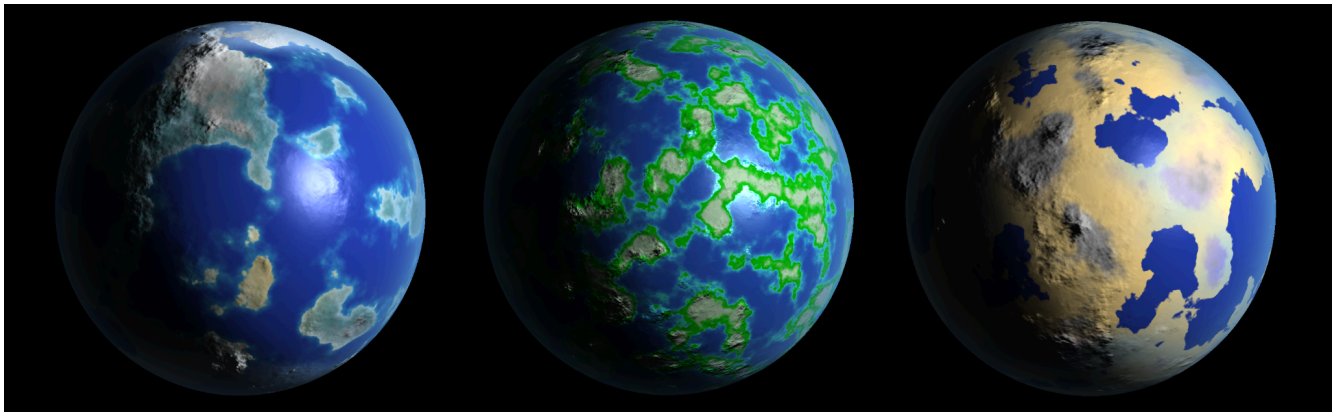

Figure 1: Three planets procedurally generated by manipulating nine variables that control, among other parameters, the planet's biomass, temperature and sunlight.

ABSTRACT

Procedural generation is a popular technique in computer graphics for generating large-scale environments. In this report, We specifically address the issue of creating planets which has several applications in industries including video games, movies, and scientific visualization. We present our approach that combines various techniques such as fractal noise, procedural normal mapping, shader programming and more. By adjusting a set of nine factors that affect the planet's size, temperature, biomass, water level, ozone thickness, rotation speed, sunlight direction, sunlight color, and its seed value for random creation, our method is able to create a broad variety of various earth-like planets. To achieve this, We start by defining the planet's geometry, then we create a fractal brownian function capable of taking 3D coordinates as inputs and outputting a noise value that maps into our planet. We use these noise values to map colors into our planet and we also generate a procedural normal map using our fractal brownian function to simulate realistic terrain. Afterwards, we calculate light reflection on our terrain by implementing a simple Phong shading algorithm with varying parameters depending on the reflected surface. Finally, we use a variety of techniques to implement our set of nine variables that influences the final appearance of the generated planet.

I. INTRODUCTION

Procedurally generated graphics is an important area of study in our field for solving problems related to generating large-scale environments. The area we are interested in, is the automatic generation of realistic and aesthetically pleasing planetary environments. Procedurally generated planets have a wide range of applications, from simulating the surface of a planet for scientific purposes, to creating entire fictional worlds for video games or movies. We present our approach to procedurally generate realistic and visually appealing planets which combines various techniques such as fractal noise, procedural normal mapping, shader programming, and more. Our system is highly customizable, allowing users to adjust a set of nine variables that influence the planet's size, temperature, biomass, water level, ozone thickness, rotation speed, sunlight direction, sunlight color, and its seed value for random generation. Our approach allows us to create a wide range of earth-like planets with different surface features, such as mountains, craters, and oceans. The resulting planets are visually striking and highly customizable.

The rest of this report is organized as follows. In Section 2, we provide a brief overview of background concepts that are relevant to our implementation. In Section 3, we describe our approach in detail, including the techniques and algorithms we used, in a step by step fashion. In Section 4, we review our results by comparing them with similar projects. Finally, in Section 5, we conclude with a summary of our findings and discuss possible future work.

II. BACKGROUND

This section provides an overview of the different tools and techniques used in the development of this project. We will explain fragment shader programming, Simplex noise, Fractal Brownian Motion (FBM), Phong shading and normal maps.

2.1 Fragment Shader Programming

This project is written in the OpenGL Shading Language (GLSL). The entirety of its code is written in the fragment shader portion.

Fragment shaders are programmable graphics processing unit (GPU) programs that are executed once for every pixel or fragment that needs to be drawn on the screen. In the context of the OpenGL pipeline, a fragment shader, calculated after the vertex-based shader, is responsible for determining the final color and other properties of each individual pixel, based on various inputs defined by the user, the most commons being a surface's normal direction, or texture coordinates. In the OpenGL fragment shader's programming language, GLSL, fragments are written as functions that take in a set of input variables and output another set of values, such as the fragment color, depth, and stencil value. They can be used to perform a wide range of operations on each pixel, including texture mapping, lighting calculations, and post-processing effects. Fragment shader programming applications are vast, we can essentially use them to do everything in computer graphics. We can, for example, do texture mapping by applying a pattern or image onto a surface, or perform lighting calculations to determine the color of a surface based on the properties of the material.

2.2 Simplex Noise

Simplex noise is a type of gradient noise commonly used in procedural generation techniques to create natural-looking patterns and textures. It was developed by Ken Perlin in 2001 as an improvement over Perlin noise. It is based on a geometrical construct known as a simplex, which is a generalization of the notion of a triangle or tetrahedron to arbitrary dimensions. The simplex noise function works on a lattice of simplex cells in multiple dimensions, where each cell contains a gradient vector pointing in a random direction. The noise function takes a set of input coordinates and finds the simplex cell containing those inputs. It then calculates the dot product between the gradient vector at the corners of the simplex cell and the distance vectors from those corners to the input coordinates. These dot products are then interpolated to generate a smooth output value. In the end, a simplex noise function takes a coordinate and returns a floating-point noise value between -1 and 1 inclusively, where coordinates that are close to each other have similar values. Simplex noise has several advantages over other noise implementation techniques, including better variation quality and faster computation speed. It has since become a popular tool in computer graphics, especially in procedural generation for terrain, textures, and the simulation of natural phenomenons.

2.3 Fractal Brownian Motion

Fractal Brownian Motion (FBM) is a multi-layered noise function. an FBM is created by adding different iterations of noise with different amplitudes and frequencies. Each layer of noise is added to the previous layer, with the amplitude and frequency of the noise increasing at each subsequent layer. The sum of these layers creates a fractal pattern, with increasingly fine-grained detail as the number of layers increases. At its most basic form, an FBM takes 7 inputs : the number of octaves, the initial frequency, the initial amplitude, the lacunarity, the gain, the noise function and the position.

The number of octaves determines the number of iterations of noise we go through to generate our final output. Frequency and amplitude are two characteristics of waves. Frequency describes the number of waves that pass a fixed point in unit time, while amplitude describes the height of the wave. Lacunarity dictates how much we increment our frequency at each iteration. Similarly, gain dictates how much we increment our amplitude at each iteration. Finally, we have the noise function that takes the position multiplied by the frequency at each iteration and returns a pseudo-random number. In the end, because an FBM uses noise functions for random generation, close position values will have relatively close output values, simulating organic-like randomness.

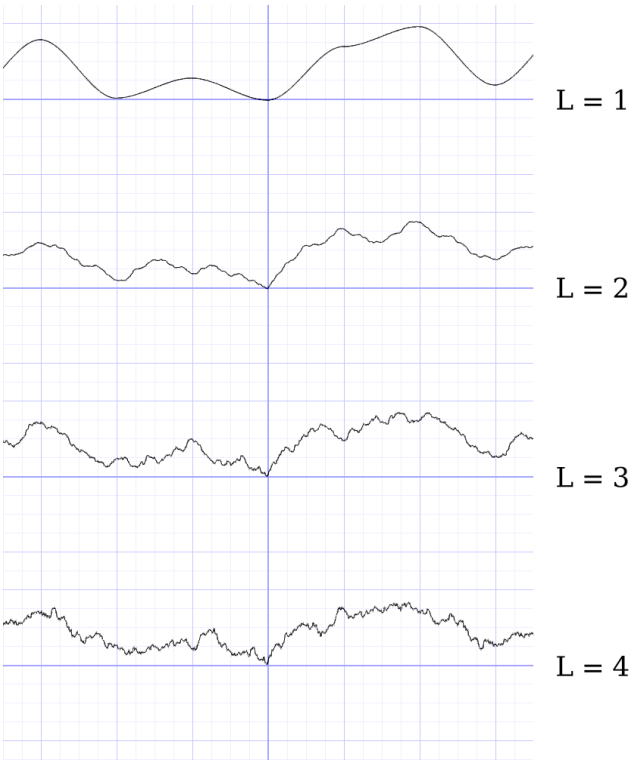

Figure 2 : An FBM curve for different values of L (Lacunarity) at ten octaves of noise. *["The Book of Shaders: Fractal Brownian Motion"]*

In the context of computer graphics, FBMs are often used to operate noise functions in order to implement procedural generation with a wide variety of inputs to manipulate.

2.4 Phong Shading

Phong shading is a shading algorithm that computes a shaded surface on the color and illumination of each pixel using the Phong reflection model seen below.

$$I_a k_a + \sum_p I_p (k_d (n \cdot l_p) + k_s (r_p \cdot v)^n) \quad (1)$$

Where I_a is the intensity of the ambient light multiplied by k_a, its ambient reflection coefficient. p is the number of light sources while I_p is the intensity of the light source. Diffuse lighting is calculated with $k_d(n \cdot l_p)$, where k_d is the diffuse coefficient multiplied by the dot product of the surface normal n and the incoming light direction l_p. Specular lighting is calculated with $k_s(r_p \cdot v)^n$, where k_s is the specular coefficient multiplied by the dot product of the direction of the reflected light r_p and the view direction v. Small n is the shininess exponent, a constant that determines the size of the specular highlight.

The algorithm simulates the behavior of light on a 3D object by approximating the ambient, diffuse and specular highlights of a surface. Phong shading uses the interpolated normal vector of a polygon's vertices to calculate the intensity and color of the light reflected at each pixel on the surface. If the coefficients are correctly adjusted, the final result is a realistic illumination effect fast enough to be usable in real time 3D graphics.

2.5 Normal Maps

Normal maps (also called bump maps) are a data structure used to simulate small surface details on 3D objects without having to add additional geometry to the model. The technique works by encoding the surface normal direction of a higher detailed 3D model on a normal map data structure, often encoded as a 2D image. This 2D image is then used as a texture map on a low-polygon version of the 3D model to give the illusion of additional detail. Normal maps can be created by using specialized software that converts high-polygon models into normal maps or algorithms that procedurally generate normals onto a surface. The resulting modified surface normal is then used in the lighting calculations, making the low-polygon model appear to have the same surface details as a high-polygon model. The illusion works because by modifying the normal vectors of a surface, we are able to simulate little bumps on an object by influencing its lighting calculations when our lighting function compares light's incoming direction with its angle of incidence which is defined by the perceived surface normal.

III. MODELING

This section provides a step by step explanation of our implementation. The equations and algorithms provided may not always be exactly the same as seen in our actual code, but they present our methodology more clearly.

3.1 Planet Geometry

In order to generate the geometry of a planet, which is essentially a sphere, we define a simple circle function that takes in a 2D position and the planet's radius and then returns the signed distance from the point to the circle with the given radius. The function's equation can be seen below:

$$d = |\vec{p}| - r \tag{2}$$

Where $\vec{p}$ is the 2D position (which is the pixel coordinate on screen) treated as a vector from the origin of the coordinate system and r is the circle's radius. subtracting the length of the pixel's vector position with the circle's radius gives us the distance d from the pixel to the circle's outline. If d is negative, it signifies that our pixel is inside our circle, while if d is positive, it signifies our pixel is outside our circle. By defining such a function we can then ignore any pixel outside of our planet's perimeter and perform computations only on the pixels inside our circle. The first computation we do if d is negative, is creating our 3D coordinates for our planet using the cartesian equation for a sphere. First, we convert the pixel coordinates to normalized coordinates by dividing them by the planet's radius, giving us the x and y position, then we find z by calculating the length of their normalized 2D position. In the end, x y and z are found this way, where p_x and p_y are the pixel position and r is the circle's (or planet's) radius:

$$\begin{aligned} x &= \frac{p_x}{r} \\ y &= \frac{p_y}{r} \\ z &= \sqrt{1 - x^2 - y^2} \end{aligned} \tag{3}$$

Having these 3D coordinates will allow us the freedom to perform familiar three dimensional calculations on our planet. One of these calculations is the rotation of our sphere to simulate planetary revolution.

3.2 Noise Manipulation

We use a Fractal Brownian Motion function to manipulate our Simplex noise function. Our implementation of the FBM is fairly conventional and takes the same inputs a regular FBM would, which include the octaves, amplitude, frequency, lacunarity and gain. Like a regular FBM, our function loops through each octave calculating a noise value starting with an initial frequency and amplitude. The frequency and amplitude are then multiplied for each octave by the lacunarity and gain parameters. The noise value is generated using a Simplex noise function that takes a 3D coordinate for input multiplied by the current frequency.

After all the octaves have been calculated, the total is divided by a factor and raised to the power of e, scaling it so that it ranges from 0 to 1. This scaling step makes mapping techniques more intuitive, because we deal with a simple range ensuring that the output of the function can be easily used as a heightmap for terrain generation and so that we can manipulate e for later use which we will talk about in the next section. The layout of our final FBM function is stated in Algorithm 1.

Algorithm 1 FBM implementation

$y \leftarrow 0$
for i = 0, i < octaves, i++ **do**
  $y \leftarrow amplitude \times SimplexNoise(p \times frequency)$
  $frequency \leftarrow frequency \times lacunarity$
  $amplitude \leftarrow amplitude \times gain$
end for
return $(\frac{y}{4})^e$

After implementing our FBM and feeding it the 3D coordinates of our planet, we can then manipulate the FBM's parameters to arrive at a satisfactory noise layout. By mapping our noise values as grayscale colors, we obtain Figure 3 below.

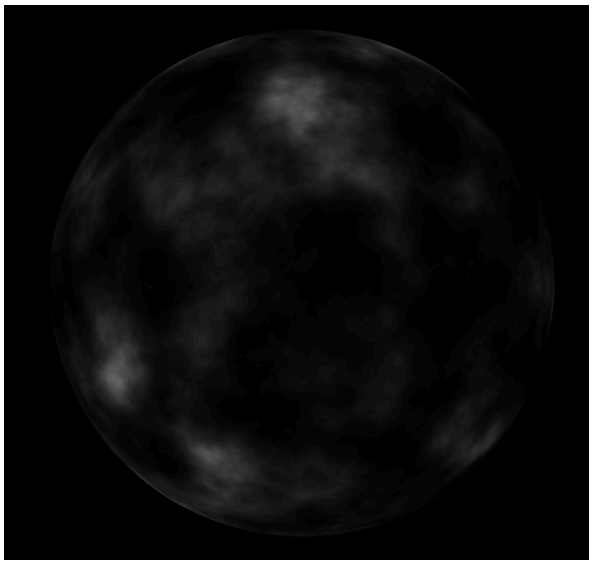

Figure 3 : The planet's noise value shown in values of gray, after being manipulated by our FBM with an octave of 6, frequency of 0.6, amplitude of 1, lacunarity of 2, gain of 0.5 and e of 4

The final step in our noise manipulation section, is to use our shader's built-in sigmoid-like interpolation on our noise to map out our land and water zones for our planet. In GLSL, there is a function called *smoothstep* that interpolates smoothly between two values based on a third value. It takes three arguments called *edge0*, *edge1*, and *x*. *edge0* and *edge1* define the start and end points of the interpolation range, while *x* is the value to be interpolated. Below is the equation of the smoothstep function, for *edge0 = 0* and *edge1 = 1* :

$$\text{smoothstep}(x) = S_1(x) = \begin{cases} 0 & x \leq 0 \\ 3x^2 - 2x^3 & 0 \leq x \leq 1 \\ 1 & 1 \leq x \end{cases} \quad (4)$$

On our implementation, we define white zones to be land and black zones to be water. However, the transition between the two color tones is too smooth as shown in Figure 3. To make their demarcation sharper, we define our smoothstep function for our land to be *edge0 = 0.05 and edge1 = 0.1*. For our water, we take higher values : *edge0 = 0.02 and edge1 = 0.08*. The *x* value for our smoothstep functions is, of course, the noise value returned by our FBM. Finally, we perform a simple linear interpolation between our land and water zones using the *mix* function in GLSL. Notice that our edge inputs in the smoothstep functions are relatively close together, this will make the output for a given range of x change more rapidly because of our close starting and ending points. This will result in sharper changes for our final noise value and the land/water demarcation will become much sharper than it was before. Figure 4 shows the final result of our noise manipulation.

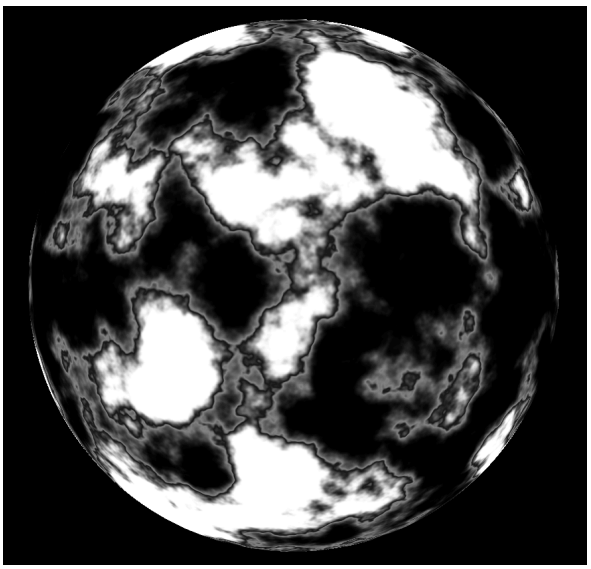

Figure 4 : The final result of our noise value after manipulating our FBM output with polynomial and linear interpolations.

3.3 Color Mapping

After having successfully separated land areas from water areas, It becomes trivial to map a green color to the white zones and a blue color to the black zones to have the basic layout of an earth-like planet. However, Only using these two colors makes the planet look very plain and artificial.

The first thing we do to enhance our planet's realism is add color variation to our land and water. To do this, we use a linear interpolation between two colors using the output of the zone's smoothstep function as the *x* value. This will make it so we can have two shades of green for our land. The more vibrant green will represent dense greenery at the center of land zones, while the less vibrant green will be at the outer layer of the land zones, where the white starts to dissipate. We do similarly for water. This will give us a smoother transition between land and water, without explicitly breaking their clear demarcation. But at this point, we are technically still using just two colors, only with different shades. Our planet still looks very artificial.

If we look at pictures of the earth for reference, biomes and mountainous areas of different colors can be seen from space. The most notable biome we can see from space are deserts. We therefore add a desert biome by creating a new FBM with new parameters for input. The parameters we use for the generation of our desert are extremely similar to the ones we used for our land and water zones, the only difference is that this time we use a lower *e* value to scale down its output value so that when we use our interpolation functions later to map the desert's FBM output, they will appear with less frequency than regular greenery. Desert frequency can, of course, be influenced by our global variables which we will talk about later. Another feature we add to our land is mountainous terrain, which we color as gray. This terrain is simply a mapping of higher FBM output values of the land zones. In other words, the whitest parts of our land will be mountainous. Another final color we add is snow and ice for the planet's poles, which does not depend on any noise coordinate. Instead, we simply map white color on our land and pale blue color to our

water solely depending on the absolute value of the planet's height, which is its y coordinate. After mixing (i.e linear interpolating) the desert, the mountainous terrain and white snow to our land, and ice to our water, then finally combining both land and water together with one last linear interpolation, we obtain the final result shown in Figure 5.

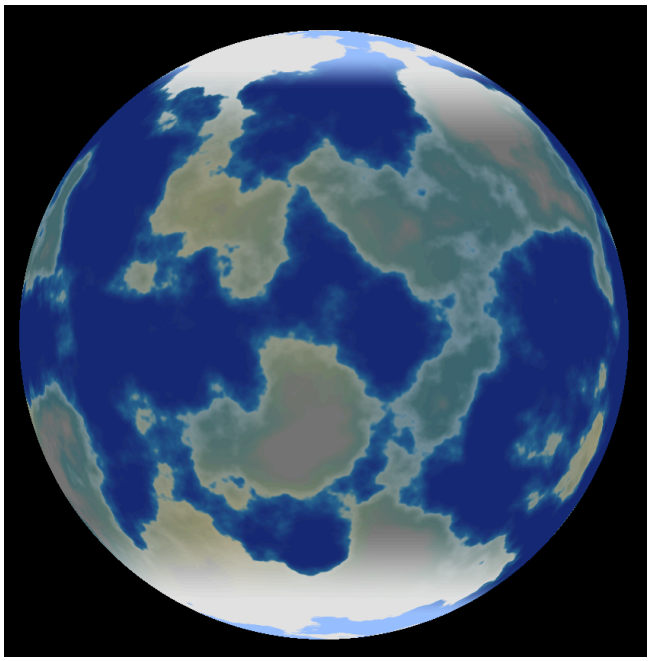

Figure 5 : The result of carefully applying all our color mapping to our manipulated noise. Color mapping takes extensive experimentation to get to a satisfactory result.

3.4 Procedural Normal Map

In order to simulate surface micro-detail on our planet, we procedurally generate a normal map on the surface of our planet by using noise to perturb our sphere's normals. Our procedural normal map function takes in 3 inputs : the planet's world coordinates, the planet's surface normals and its FBM noise output value.

The function first calculates a noised position value. This is a position that has slightly been jittered away from its original position on the sphere. We generate this "jitter" by creating a variation vector with the amount of distance we want to jitter our original position. We then use this vector alongside our position as new inputs to our FBM differentiation function that calculates a new randomly jittered position. This function calculates the difference of our FBM output by subtracting an FBM with the position *plus* the jitter amount, against the position *minus* the jitter amount. This gives us a noised position in which we can calculate a new normal on top off, which will be slightly different from the original normal our fragment had before. This difference between normals will be important to simulate surface detail when applying our Phong lighting later in the next section.

After we have our jittered positions, we calculate a normal depending on the region our fragment is in. Indeed, because water is flatter than land, and land is flatter than mountainous regions, we want to have a different amount of perturbation for each of these regions. Preferably, we want our water to appear less bumped than our land and our land less wrinkly than our mountains. To do this we multiply our jittered position by a constant amount depending if it's water land or mountains. Water will have a small amount of perturbation, while mountains will have a big amount because we want them to appear tall and textured when we implement our lighting. Below is a suggested simple layout to implement our procedural normal map algorithm.

Algorithm 2 Procedural Normal Map

```
variation ← 0.01
newPosition ← FBMdifference(position, variation)
if region == water then
    constant ← 1
else
    if region == land then
        constant ← 6
    else
        constant ← 10
    end if
end if
newNormal ← normalize(constant × newPosition + normal)
return newNormal
```

3.5 Phong Implementation

Our Phong shading implementation is conventional. Our lighting function takes as input the fragment's normal direction which has been modified by our procedural normal mapping and uses it to calculate diffuse and specular lighting. For ambient lighting, we only use a constant. Our implementation of Phong shading is simply a direct implementation of Phong's reflection model seen in Equation 1. The only important modification we do is that we treat specular reflection differently for land and water. Because water is supposed to be much more specular than ground, we apply a higher specularity coefficient if the fragment we are performing calculations for is located in water. Our final Phong shading result is a single floating point number representing light intensity, which we then multiply to our planet's color to give the final shaded result as seen in Figure 1. Notice how specularity varies a lot depending if it is on water or on land. Also notice how terrain seems much rougher on mountainous areas. This is because of our procedural normal mapping function which gives bigger normal variations on mountains than on land or water. As seen in the background section on Phong shading, diffuse light is the dot product of the surface's normal and the light direction, and specularity uses the reflection vector which needs the surface's normal too. The perturbation of our normal direction is the reason why we are able to affect our shading so much to make our planet appear textured.

3.6 Aesthetical elements

Our final results would not appear as great if we didn't add small aesthetical features to make our planet look better. This section is mostly to cover some small but important additions to our project. One of the things we do to aesthetically improve our planet generation is add its planetary rotation. Our fragment shader takes in a time uniform, which we then use to calculate rotation over time. We implemented a rotation function which contains a basic y rotation matrix that takes in time as input. We then multiply the result of our rotation matrix to our world position, giving it new coordinates every frame. As a result, we obtain a rotating planet, making it look much more

realistic. Another feature we added, this one much more subtle, is the ozone layer effect. This is not at all scientifically accurate because we don't know what visual effects are caused by the ozone layer, but nonetheless, all along the edges of our planet, we added a very faint blue gradient to give it a special ozone effect. This is done by calculating a gradual blue transition over the z axis, which is our deepness axis, using a smoothstep interpolation. We do not apply phong shading on our ozone effect to make it glow, for stylistic reasons.

3.7 Global Variables

We implemented a total of 9 global variables able to manipulate the appearance of our planet. Take note that in this section, we explain how we implemented seemingly realistic variables that affect a planet's appearance, but none of them is truly scientifically accurate, as the end goal is to have simple to manipulate variables and an aesthetically pleasing planet. Will will briefly explain each variable without going into too much detail.

The first global variable is called *Seed*, which is a common term to describe a number that initializes a random function. The seed value is injected into our simplex noise inputs inside our FBM function. Changing the seed will completely randomize everything that is procedurally generated on our planet. The next variable is called *Temperature*. As the name suggests, this variable changes the average temperature of our planet. This one simply changes how much of our planet's surface does the snow and ice take in the north and south poles. We simply add our Temperature value to the edge inputs of our smoothstep function that color maps snow and ice. A temperature with a value equal to one removes the snow of the north and south poles completely, while a value of minus 1 covers the entire planet in snow and ice, making it look like a very cold planet (Figure 7).

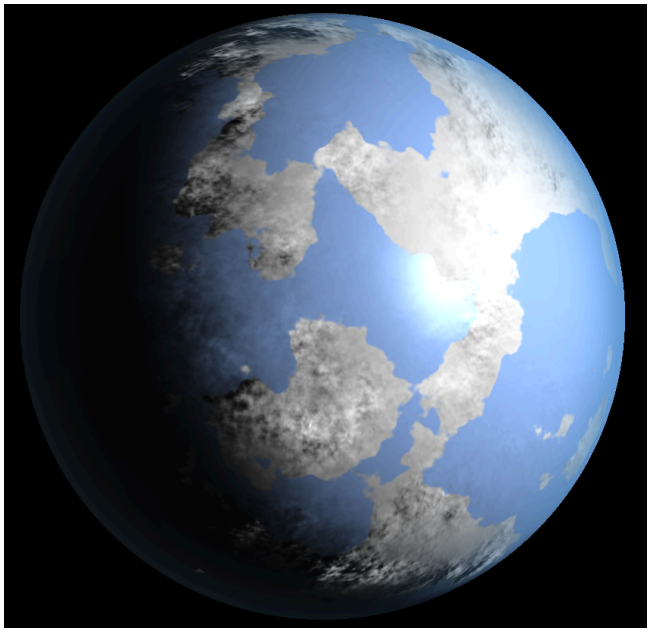

Figure 7 : A planet with its temperature value set to -1

The next variable is called *Biomass*, which is a term used in ecology to measure the amount of biologic material in a region. Usually, the more biomass a planetary region has, the more dense its forests are. We simulate this by increasing the amount of greenery our planet has by injecting the value of the Biomass variable in the coloration of our land by multiplying the amount of green in our RGB by the Biomass value in our interpolation function. We also multiply our Biomass with the parameters of our desert FBM to control its frequency, this way, with a very high Biomass value will eliminate desert from our planet, while a very low biomass value will transform our planet into a fully desert planet. Observe the rightmost planet in Figure 1 to see what a low biomass value looks like. The next variable is called *Water Level,* which controls the amount of water our planet has. Recall how we mapped land and water on our planet. What this global variable does is increase the amount of "black" zones it accepts into the water zone. This variable multiplies the edge values of our interpolation function to influence the amount of water that will be mapped in our noise function. A high water level will transform our planet into what astronomers call an *ocean World* while a low amount of water level will eliminate water from our planet. The next global variable is called *Ozone* which controls the amount of ozone effect on our planet. This function simply multiplies the ozone's color values and the edge values of its interpolation function. The next variable is called *Planet size* which determines the radius of our planet, nothing else. Another variable is called *Rotation*, which dictates our planet's rotation speed by multiplying the time value inside our rotation function. The last two variables are called *Sunlight* and *Sun Position* which affects our Phong shading algorithm. These are quite literally variables in phong's reflection model. Sunlight is the light's intensity (I_p). The intensity is a 3d vector, so it includes the sun's color which can be modified, while Sun Position is transformed into a direction to become light's direction (l_p) in our equation.

These nine variables can be manipulated independently from each other to produce a very large variety of planets. We suggest the reader try them out to produce new exotic looking planets.

IV. RESULTS

In this section we compare our work with similar projects and talk about the performance of our implementation.

4.1 Comparison with NASA's Blue Marble

While creating our planet, we were first modeling it to resemble earth using this reference 3D model made by NASA (figure 8), in order to make our planet look as close to a real planet as possible. This model was made based on the Blue Marble photograph taken by Apollo 17. Our attempt at making an earth-resembling planet can be seen by looking at the leftmost planet in Figure 1. Our humble attempt is far from being as realistic as NASA's 3D model, the most obvious difference being our lack of clouds. The second most notable difference is the land, and especially the water's coloration. We are proud, however, to have adjusted our shading just right to make it almost identical to NASA's implementation. By observing their model rotating on itself, it is possible to notice that the specularity of their planet is very close to ours, as well as their diffuse and ambient lighting. Another thing worth noting is that they seem to have implemented some sort of anti-aliasing to the extremities of their planet, while on our planets, aliasing can be easily noticed, especially on low resolution devices.

Figure 8 : Nasa's 3D reproduction of The Blue Marble *[*"*Elegant Figures - Crafting the Blue Marble*"*]*

4.2 Comparison with Space Engine

Space Engine is a proprietary software developed by Cosmographic Software, founded by the astronomer and programmer Vladimir Romanyuk. It is a 3D astronomical visualization software that allows users to explore and navigate the universe in real-time, with a focus on scientific accuracy and attention to detail. Space Engine allows users to explore hypothetical scenarios, such as traveling to distant star systems and to customize celestial objects such as planets (Figure 9). This software makes heavy use of realistic procedural generation based on real scientific data, which results in a very realistic simulation of space. Space Engine's procedural planet generation is extremely complex and allows users to manipulate around two hundred parameters! Compared to our procedural generation with only nine global variables to manipulate, it is quite an enormous difference. Their procedural generation produces very realistic planets with the ability to travel through their land and atmosphere. Our project only generates the outer visuals of a planet, it would be impossible with our implementation to travel "inside" our planets.

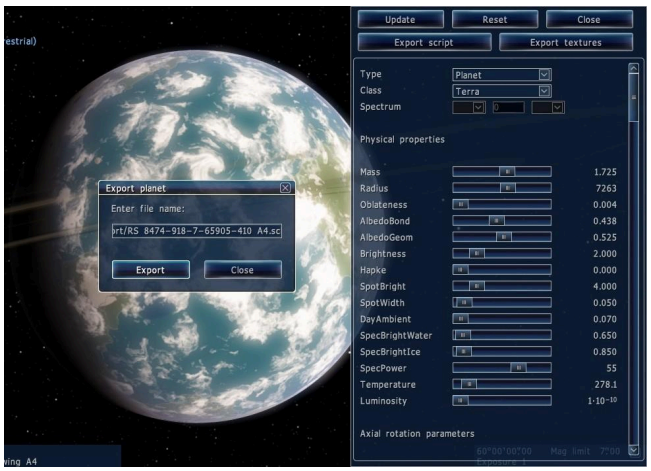

Figure 9 : Space Engine's planet editor *[*"Creating a planet – Space Engine*"]*

Nevertheless, it is worth mentioning that Space Engine was the main inspiration for this project. It's only while developing procedural computer graphics that it becomes apparent how difficult it is to manipulate random noise functions to produce realistic procedural generation, and Space Engine got me impressed with their results.

4.3 Performance

By running our shader through WebGL with Three.js on the Chromium web browser, we can observe a stable performance of 80 to 90 frames per second on a monitor with 1920 pixels horizontally and 1080 pixels vertically. We are running our program on a Linux machine with a GeForce GTX 1650 graphics processing unit, which is a mid-range GPU used for handling intermediate computer graphics tasks. We believe this performance is adequate, considering that no FPS stuttering occurs and that it is running through WebGL instead of regular OpenGL.

V. CONCLUSION

In this report, we presented our approach to generate planets procedurally. We worked entirely in GLSL, in the fragment shaders portion, but by defining our planet's geometry we were able to work in 3 dimensions. We then created our FBM function and found the right parameters to map our noise function into our sphere. Afterwards we extensively experimented with mapping our colors into our noise values to produce a realistic-looking planet. Finally, we implemented phong shading alongside procedural normal mapping to simulate micro-details on our planet.

We learned a lot in the making of this project, especially on the topic of procedural generation. We spent a large amount of time reading and learning on the subject, in fact, a lot of what we learned couldn't be shown here on our work because it wasn't particularly useful for our specific problem. An example of that is the extensive reading we did on procedural signed distance function techniques. But the only thing remotely close we have to that is our circle SDF for our planet.

We plan to keep working on this project. One thing we want to implement is the procedural generation of clouds, which is the most important detail that's missing to achieve making a believable planet. Another thing we want to add is more parameters to customize our planet, notably, we want to add the ability to control the amount of continents our planet has. Also, we want to add an user interface to control these parameters, instead of changing them directly in the GLSL code.

REFERENCES

Fragment Shader - OpenGL Wiki, 25 November 2020, https://www.khronos.org/opengl/wiki/Fragment_Shader.

L*ecture 17 - more on Texture mapping - procedural shading*, https://www.cim.mcgill.ca/~langer/557/17-slides.pdf.

"The Book of Shaders: Fractal Brownian Motion." *Book of Shaders*, https://thebookofshaders.com/13/.

"Creating a planet – Space Engine." *Space Engine*, https://spaceengine.org/manual/making-addons/creating-a-planet. Flick, Jasper.

"Simplex Noise." *Catlike Coding*, 23 September 2021, https://catlikecoding.com/unity/tutorials/pseudorandom-noise/simplex-noise/.

"GLSL Shaders - Game development | MDN." *MDN Web Docs*, 23 February 2023, https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_on_the_web/GLSL_Shaders.

"Procedural Noise: Value and Gradient Noise in GLSL [Shaders Monthly #8]" *YouTube*, 4 July 2022, https://www.youtube.com/watch?v=jkYIOu8HddA.

Guo, Xichun Jennifer, and Bruce Land. "Phong Shading and Gouraud Shading." *Cornell ECE*, https://people.ece.cornell.edu/land/OldStudentProjects/cs490-95to96/GUO/report.html.

Simondev, "GLSL course" Shader Online *Class* https://simondev.teachable.com/p/glsl-shaders-from-scratch

Gustavson, Stefan. "(PDF) Simplex noise demystified." *ResearchGate*, 1 january 2005, https://www.researchgate.net/publication/216813608_Simplex_noise_demystified.

"LearnOpenGL - Normal Mapping." *Learn OpenGL*, https://learnopengl.com/Advanced-Lighting/Normal-Mapping.

"PROCEDURAL PLANET GENERATION IN GAME DEVELOP- MENT." *Theseus*, https://www.theseus.fi/bitstream/handle/10024/82203/opinnaytetyo.pdf?sequence=1&isAllowed=y.

"Recomputing normals for displacement and bump mapping, procedural style." *GitHub Pages*, https://stegu.github.io/psrdnoise/3d-tutorial/bumpmapping.pdf.

"Rotating Blue Marble." *YouTube*, 6 October 2011, https://www.youtube.com/watch?v=laiVuCmEjlg.

"Elegant Figures - Crafting the Blue Marble." *NASA Earth Observatory*, 6 October 2011, https://earthobservatory.nasa.gov/blogs/elegantfigures/2011/10/06/crafting-the-blue-marble/. Robert Simmon

"[PDF] Fractional Brownian motion in a nutshell." *Semantic Scholar*, 8 June 2014, https://www.semanticscholar.org/paper/Fractional-Brownian-motion-in-a-nutshell-Shevchenko/5609886331c316a550900bbe04c5878384f81a05. G. Shevchenko