Procedural Planet

50.017 Graphics and Visualisation Project

Team 6
Jeremy Chew 1003301

Table of Contents

Introduction

In this project, I sought to generate realistic-looking planets, together with atmospheric lighting and volumetric clouds.

I was inspired by open-world space-exploration games such as Outer Wilds[1], Kerbal Space Program[2], as well as No Man's Sky[3]. In these games, players are able to explore expansive planets and solar systems, and I wanted to be able to generate my own procedurally-generated planets.

I was also heavily inspired by Sebastian Lague's Coding Adventures series on building a Solar System[4]. In this playlist, he details his process in building a solar system from scratch in Unity. The references he listed and the code he provided proved to be immensely helpful when it came to implementing my own version of a planet generator.

Hence, the motivation behind this project was to build a system where one can render realistic-looking procedurally-generated planets. The intention behind this project was to replicate certain real-world phenomena, such as atmospheric-scattering induced sky colour, and volumetric clouds with phase lighting and scattering.

Approach

The process behind building a procedural planet generator is relatively complex. As a rough estimate, one only needs to observe that Sebastian Lague, who creates such videos as a full-time job, took about a year to produce the 4 videos in his Solar System playlist.

Given the short timeframe of this project, however, it was important to split the work into manageable chunks. Each chunk is organised such that it builds upon the previous, and is only focused on implementing a single feature at any given time.

Broadly speaking, the project was split into the following 7 parts:

1) Sphere
2) Terrain
3) Colouring
4) Ocean
5) Atmosphere
6) Clouds
7) Others

---

[1] https://store.steampowered.com/app/753640/Outer_Wilds/
[2] https://www.kerbalspaceprogram.com/
[3] https://www.nomanssky.com/
[4] https://www.youtube.com/playlist?list=PLFt_AvWsXl0cSy8fQu3cFycsOzNjF31M1

The full details for each section would be explained in greater detail under the Implementation section.

The project was also written in C++, using the GLFW[5] library for window management. GLAD[6] was used to load OpenGL, and GLM[7] was used as the library of choice for vector math. For the user interface, I decided to use DearImGui[8] as it was an extremely simple-to-use and fast library with pre-existing integrations with GLFW. For image loading, the simple stb[9] library (specifically stb_image.h) was used.

In terms of input data, most of the project used procedurally generated data generated using mathematics. However, I did use two additional textures. The first was a water normal map downloaded from an online tutorial by ThinMatrix on water[10]. The second was a ground normal map obtained by searching Google, and attributed to Eisklotz[11].

The full source code of the project can also be found at https://github.com/Mickey1356/procedural-planet.

Implementation

In this section, I will detail the main techniques and algorithms used to address each individual component of the project.

Sphere

The sphere, evidently, would be the first task to tackle in this project. It would represent the planet, and form the base on which additional details and effects could be added.

There are various algorithms used to generate a spherical triangle mesh. Perhaps the most often-used one is the UV Sphere (Figure 1).

[5] https://www.glfw.org/
[6] https://glad.dav1d.de/
[7] https://github.com/g-truc/glm
[8] https://github.com/ocornut/imgui
[9] https://github.com/nothings/stb
[10] https://www.youtube.com/playlist?list=PLRIWtICgwaX23jiqVByUs0bqhnalNTNZh
[11] https://www.eisklotz.com/products/seamless-pbr-textures/forest-ground-03/

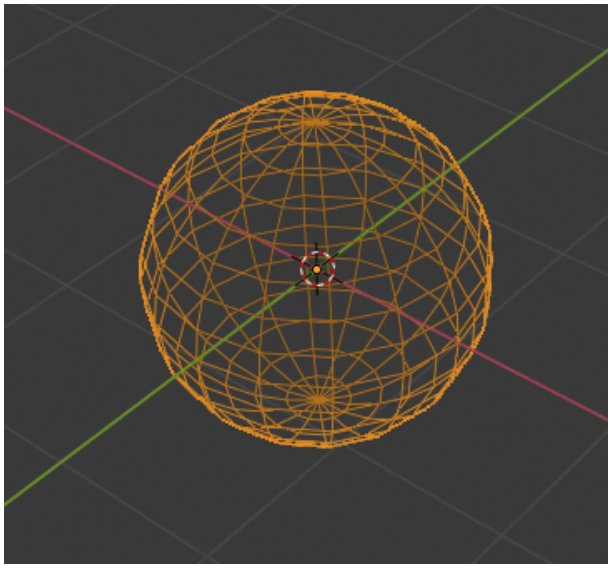

Figure 1: Wireframe of a UV Sphere

While it was relatively simple to implement, this type of sphere would not be suitable for this project as it's vertices are not very equally distributed across the entire mesh. Specifically, I would obtain a mesh with great detail at the poles only, while the equatorial regions would be less detailed. This would not be very suitable for terrain generation, where one would expect relatively equal detail all around the mesh.

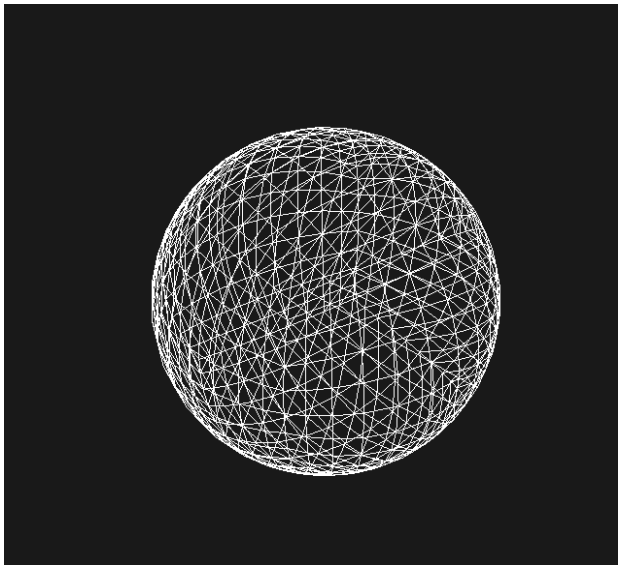

Figure 2: Wireframe of a cube-sphere

A better alternative that I used was the cube-sphere (Figure 2). This is generated by subdividing a cube and projecting its vertices into a certain radius away from the centre, resulting in a spherical mesh that has its vertices relatively evenly distributed.

Generating a triangle mesh for a cube is easy enough as one can simply generate 6 planes and align them accordingly. However, this would result in the duplicate vertices along the edges, which may cause discontinuity issues in the future. As such, special care was needed to avoid this issue and ensure that the edge vertices were properly “shared” across the involved triangles.

In order to avoid unnecessary computation time, the vertices are computed only during initialisation and when the number of subdivisions were changed. The projection of the vertices was also separately done in the vertex shader, which, while it sped up computation time, would come with certain drawbacks that will be discussed later. The affected areas were outside the scope of the project, however, so the decision was upheld.

Figure 2 above was generated as an intermediate result from this stage.

Terrain

The terrain was generated using 3D noise (code obtained online[12]). The underlying mathematical principles would not be explained here.

To convert noise into terrain is a relatively straightforward task that has been tackled by plenty of articles and papers. The rough idea would be to treat the noise value as a heightmap and offset the vertices by that amount. In terms of a sphere, the vertices would basically be projected to be at the calculated distance from the centre.

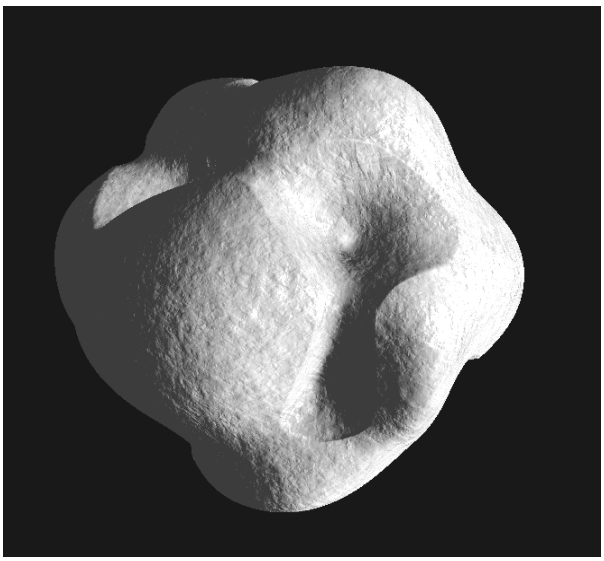

Figure 3: Noise sampled at a single frequency

Using just a single noise value results in an extremely blobby-like volume that does not resemble terrain (Figure 3). To circumvent this, I sampled more noise at increasing frequencies

[12] https://github.com/ashima/webgl-noise

and added them to the original noise value. Normally, this would result in the sphere becoming increasingly spiky (Figure 4), so each noise sample was premultiplied by a decreasing amplitude factor to reduce its effect on the final value.

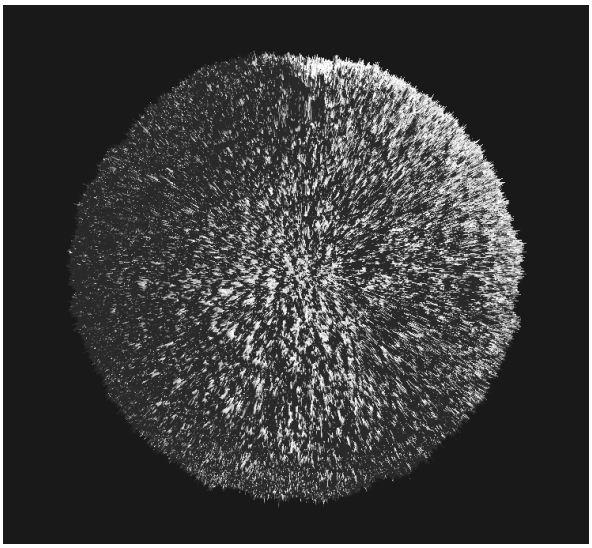

Figure 4: Spiky terrain

In more technical terms, the number of times I sampled the noise are called **octaves**, the frequency multiplier is known as **lacunarity**, and the decreasing amplitude factor is known as **persistence**. Lacunarity is usually set to a value more than 1, and the frequency is multiplied by the lacunarity every time another sample of the noise is taken. Persistence is usually set to a value smaller than 1, and the amplitude of the noise is multiplied by the persistence factor for each sample. This results in noise being sampled at exponentially increasing frequencies, by contributing at an exponentially decreasing percentage, and the entire process is known as fractal Brownian motion, or fractal noise.

Mathematically, the noise signal can be written as $height = \sum_{k} Ap^{k} noise(xSl^{k} + o)$ where k is the number of octaves, A is the initial amplitude, p is the persistence, S is the initial frequency, l is the lacunarity, and o is the offset. The offset can be used as a “seed” to generate different terrain.

The noise values were also multiplied by a small noise multiplier value, in order to maintain the original shape of the sphere. Figure 5 below was generated as a result.

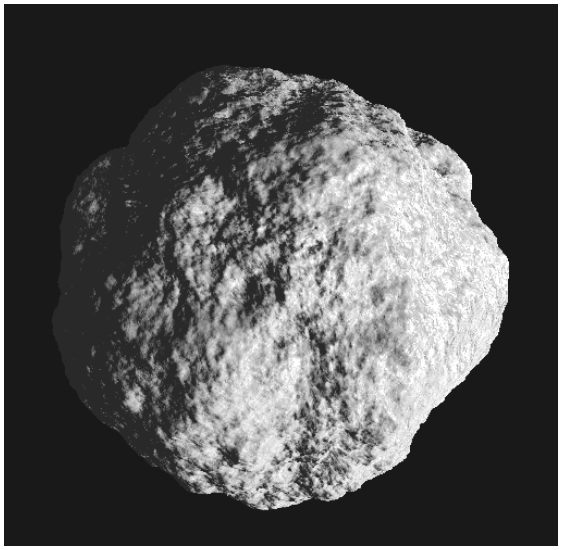

Figure 5: Terrain generated with fractal noise

The resultant terrain was too flat, so I decided to add some additional parameters to generate oceanic areas. The algorithm simply takes all values below a certain threshold, and smooths[13] them to the ocean depth. All oceanic areas were also multiplied by an ocean multiplier parameter to deepen them. This results in Figure 6 below.

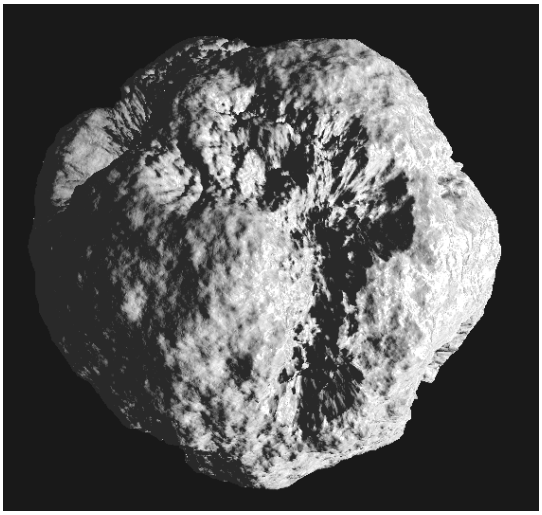

Figure 6: Final terrain with oceanic trenches

Colouring

Colouring the terrain involved both the planet's terrain colour, as well as its lighting.

[13] https://iquilezles.org/www/articles/smin/smin.htm

I used the Phong lighting model because it was simple to implement and gave relatively good looking results. Furthermore, the focus was not on physically-based lighting of the planet model, but rather the additional effects that could be added overall.

The Phong lighting model required the normal vector of each vertex. In order to calculate this, I took samples of the positions along the tangent and bitangent vector at a given point, and used the cross product to compute the normal vector. This was done in the vertex shader and passed to the fragment shader.

In order to add finer detail (as seen in Figure 3), I used a normal map downloaded from Google. However, there was still a need to generate a UV mapping between the planet mesh and the texture. I experimented with spherical UV mapping as discussed in the lectures, but results were quite poor, with obvious stretching occurring. The method I used in the end was triplanar mapping, where one projects the texture along three axes and blends them together according to the original position and normal of the vertex. I followed an online tutorial[14] that discussed several options, and used the method known as “Reoriented Normal Mapping”.

For the actual terrain colour, I used a combination of the height of the pixel (distance from centre) as well as the steepness (computed as $1 - dot(localUp, normal)$ where ***localUp*** is simply the direction from the centre to that point). The colours were separated into the grass, snow, sand, seafloor, and ground, each with their own associated threshold values for their heights and blending parameters. The grass colour had an additional pair of parameters to account for the steepness.

The blending function used was $clamp(\frac{x-h}{t-h}, 0, 1)$ where $h = t\,(1 - b)$ with *b* the blending parameter and *t* the threshold value. Intuitively, the function outputs values between 0 and 1, giving 1 when *x* is equal to *t*, and linearly increases from 0 to 1 when *x* varies from *bt* to *t*. I tuned the parameters until I achieved a result that looked realistic.

The final rendered result at this stage is shown below in Figure 7.

[14] https://bgolus.medium.com/normal-mapping-for-a-triplanar-shader-10bf39dca05a

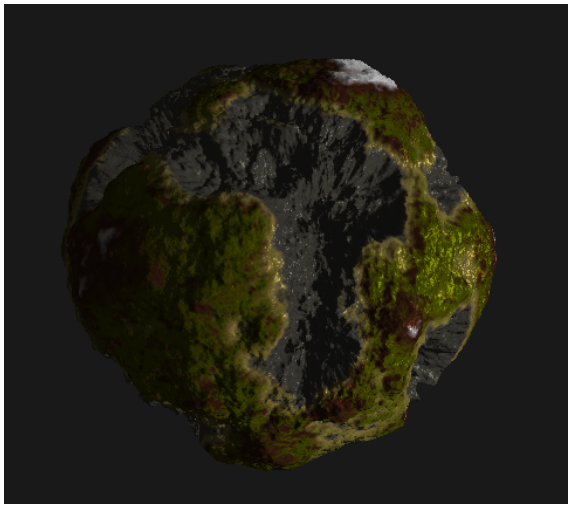

Figure 7: Coloured terrain

Ocean

From this point on, the additional effects were implemented as post-processing effects.

In order to implement post-processing effects, a framebuffer was set up such that the first pass (drawing the planet mesh and colours) could render directly to a texture. Then, a simple quad was initialised to render the texture to the screen (along with its associated vertex array object). A different pair of vertex/fragment shaders was attached to this new vertex array object, and the rest of the effects described henceforth were implemented entirely in this fragment shader.

The ocean was implemented as a sphere with a constant radius centred exactly at the planet's centre. This was done by using a ray-sphere intersection algorithm to determine the point at which the view ray intersects the sphere.

I computed the distance a given pixel is from the camera by using the depth texture. As the values stored in the depth texture were nonlinear, it was converted to distances by using the near and far planes of the camera. The equation used to convert from non-linear distances to linear is $\frac{2 * near * far}{far + near - d * (far - near)}$where $d = 2 * depth - 1$.

Then, I determine if the camera would be looking through the ocean, if it is, I draw the ocean colour to screen. As an initial test, I set the ocean colour to white, resulting in Figure 8 below.

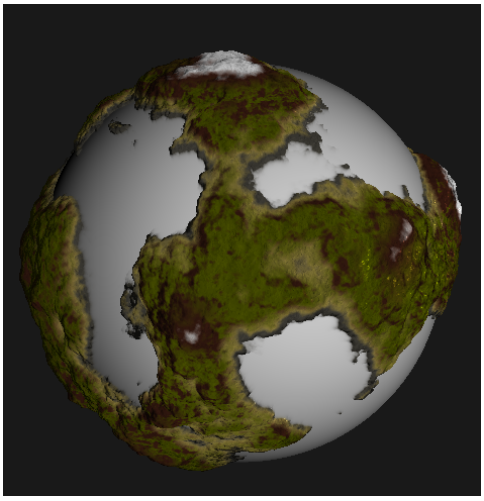

Figure 8: White oceans

To colour the ocean, I selected two colours for the shallow and deep regions respectively. Then, the two colours are linearly mixed together based on an additional parameter which uses the view depth. Finally, I introduced another parameter to tune the transparency of the water.

After diffuse lighting was also added, Figure 9 below was rendered.

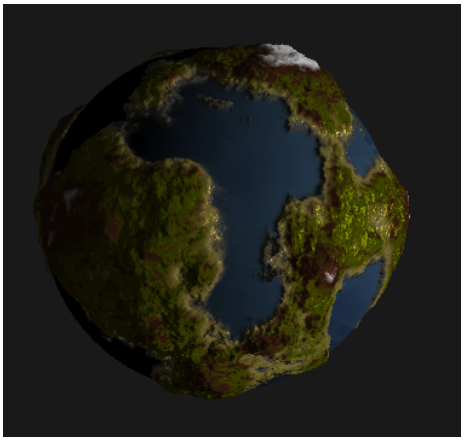

Figure 9: Blue oceans

I added detail to the ocean by introducing a water normal texture provided by an online tutorial. Similar to the terrain, triplanar normal mapping was used to properly extract the normals from the texture. They were then linearly mixed with original normals according to a parameter to simulate wave strength.

The diffuse lighting system was edited to use the new normals, and specular highlights were added in order to obtain the illusion of waves reflecting the Sun.

The texture was sampled at ever-changing offsets (as a function of the elapsed time) in order to add motion and dynamism in the scene. As the texture was loaded in with the options of

GL_REPEAT, I did not need to perform any additional calculations to keep the sampled UV coordinates in the range of [0, 1] as OpenGL would have taken care of that already.

Figure 10 shows the results at this stage.

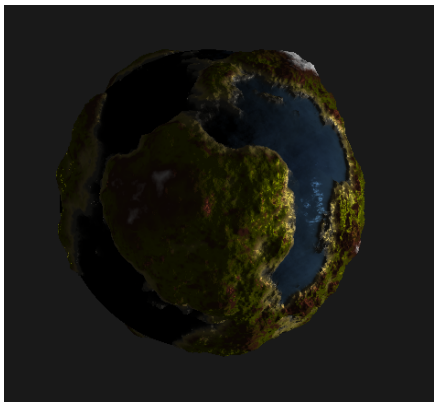

Figure 10: Normal-mapped ocean with specular highlights

Atmosphere

For the atmosphere, I referenced Sebastian Lague's code as well as the articles he had linked in the description of the video. An article from GPU Gems 2 (Chapter 16: Accurate Atmospheric Scattering[15]), also goes into great detail on how to implement atmospheric scattering in the fragment shader.

The atmosphere is similar to the ocean in that it is represented as a sphere centred exactly where the planet would be. However, instead of drawing a single colour (as in the ocean), the colour of a given pixel in the atmosphere is computed by simulating physical laws and determining how much light from the Sun reaches the camera.

This amount of light is computed by ray-marching along the view direction and sampling both the optical depth as well as the density at the given points. The density is computed as a function of the height above the surface of the planet (assuming the planet is perfectly spherical), and ranges from 0 at the top of the atmosphere to 1 at the bottom. The rate of change of the density is also controlled by a density falloff parameter. The exact function used was $density = exp(-ht01 * falloff) * (1 - ht01)$ where $ht01 = \frac{height}{r_{atmosphere} - r_{earth}}$

The optical depth[16] at a given point measures the ratio of incident light to transmitted light. Instead of computing it exactly, I approximated this value by ray-marching towards the light and taking the average of the densities at each point. This gave the optical depth from the Sun to the

---

15 https://developer.nvidia.com/gpugems/gpugems2/part-ii-shading-lighting-and-shadows/chapter-16-accurate-atmospheric-scattering

16 https://en.wikipedia.org/wiki/Optical_depth

point in question. Then, I also computed the optical depth from the point to the camera in the exact same manner.

The light that reaches a certain point is calculated as:

$inlight = density * transmittance * stepsize$ where
$transmittance = exp(-(od_{sun} + od_{view}))$.

The step size is dependent on how many sampled points were taken, which was a global parameter that could be changed in the editor.

Once this is complete, the following figure (Figure 11) was able to be rendered.

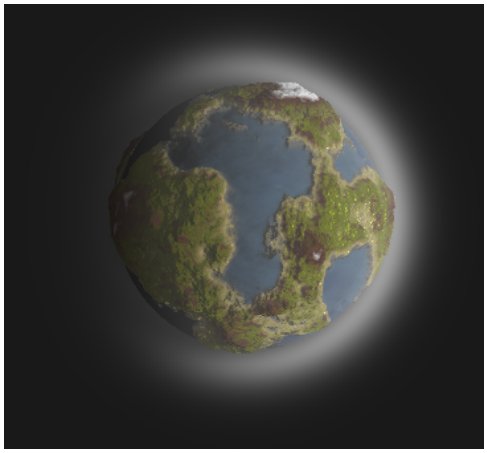

Figure 11: Plain atmosphere

In the article, the optical depth is computed by pre-baking a lookup texture. However, I did not take that approach as I wanted to focus on simplicity and not worry about pre-baking and generating textures. Fortunately, performance was not an issue even without this optimisation, and the application was still running at a stable 60 fps.

To complete the atmosphere, I added colour. For Rayleigh scattering (which is used when the scattering particle is smaller than the wavelength of light, as in the case of the atmosphere), light is scattered proportionate to the inverse of the fourth power of its wavelength[17]. As computers use RGB, the physical wavelengths of red, green, and blue lights were recorded[18] and used. The original light calculation function was modified to include the scattering values of the lights, resulting in Figure 12 below. More specifically, the incoming light was modified to $inlight = density * transmittance * stepsize * scatter$, and the transmittance was modified to $transmittance = exp(-(od_{sun} + od_{view}) * scatter)$.

[17] https://en.wikipedia.org/wiki/Rayleigh_scattering
[18] https://en.wikipedia.org/wiki/Spectral_color

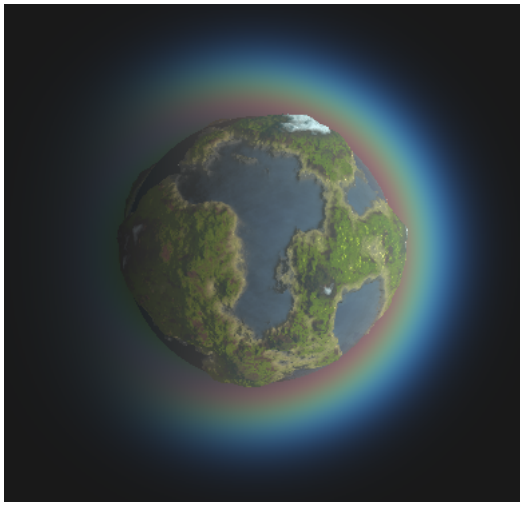

Figure 12: Coloured atmosphere

Clouds

For clouds, the main resources used were a set of slides[19] presented by Andrew Schneider, Principal FX Artist for the game Horizon Zero Dawn, as well as a talk[20] on Youtube presented by Julian Oberbeck. The two resources discuss the same set of ideas, with the talk essentially being the speaker's implementation of the system presented in the slides.

Nevertheless, the main idea behind the two resources is to represent the density of clouds as a combination of Worley[21] and Perlin[22] noise. Then, one uses a ray-marching approach akin to computing the atmosphere in order to calculate how much light reaches a sampled point.

In my implementation, due to the lack of time, I decided not to pre-bake the noise textures and instead used fractal noise similar to how the terrain was generated. This time, however, the performance took a more significant hit, and the frames per second dropped to around 45 or so when there was larger cloud coverage. Nevertheless, I considered it to be relatively respectable, and the benefit of avoiding loading flip book textures in OpenGL was a worthy tradeoff.

The offset in the noise equation was also written as a function of time ($o = speed * time$) in order to obtain animated clouds.

The cloud layer is represented as a sphere similar to the ocean and atmosphere. This time, I included an additional radius representing the minimum height at which clouds would start to appear. In order to have a smooth transition from "no clouds" to "clouds", a height signal was

---

19 http://advances.realtimerendering.com/s2015/The%20Real-time%20Volumetric%20Cloudscapes%20of%20Horizon%20-%20Zero%20Dawn%20-%20ARTR.pdf

20 https://www.youtube.com/watch?v=8OrvIQUFptA

21 https://en.wikipedia.org/wiki/Worley_noise

22 https://en.wikipedia.org/wiki/Perlin_noise

computed according to another paper[23], and the sampled density at a given point would be multiplied by this height signal. The height signal function was $signal = 1 - h^2$ where $h = 2 * \frac{clamp(height - r_{min}, 0, r_{max} - r_{min})}{r_{max} - r_{min}} - 1$. This results in a quadratic curve where the maximum is exactly in the middle of the cloud layer, and decreases to 0 at both r_min and r_max.

At this point, I ray-marched the clouds (similar to how the atmosphere was ray-marched), and returned the sum of densities. The denser a point is computed to be, the whiter it would be. This resulted in the following figure (Figure 13).

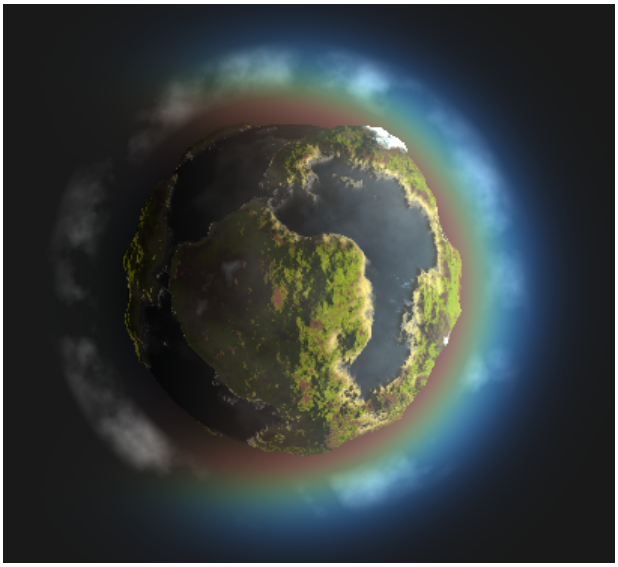

Figure 13: Basic clouds

The final step was to compute the lighting for the clouds. In order to do so, an approach extremely similar to the atmosphere was taken, where two ray-marches were performed, one from the camera (already implemented) to a given pixel in world space, and one from a pixel in world space towards the Sun (to calculate how much light would reach the given pixel).

For the second ray-march, the densities were sampled at each marched point and the averaged density was returned. Finally, for each sampled point during the first ray-march, the Henyey-Greenstein phase function[24] was used to calculate how much light would be scattered in from the Sun towards the camera. For brevity, the equation is not listed here, but it is directly dependent on a parameter *g*, and the cosine of the angle between the direction from the camera to the sampled point, and the direction from the sampled point to the light source.

---

23 https://cescg.org/wp-content/uploads/2018/04/Michelic-Real-Time-Rendering-of-Procedurally-Generated-Planets-2.pdf

24 https://www.astro.umd.edu/~jph/HG_note.pdf

Putting this all together, I rendered the following image (Figure 14), where one can see how the Sun would light up clouds that are in front of it.

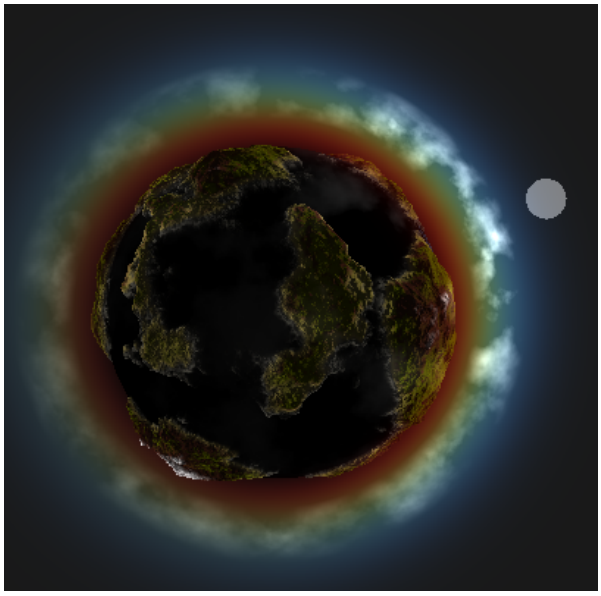

Figure 14: Lit clouds

Others

In this section, I detail several other features that did not fall in either of the categories above.

User Interface

As mentioned previously, the user interface was built using Dear ImGui. I wanted to be able to control every single aspect of the procedural generation, so for every controllable parameter, I included it within the parameter editor interface (Figure 15).

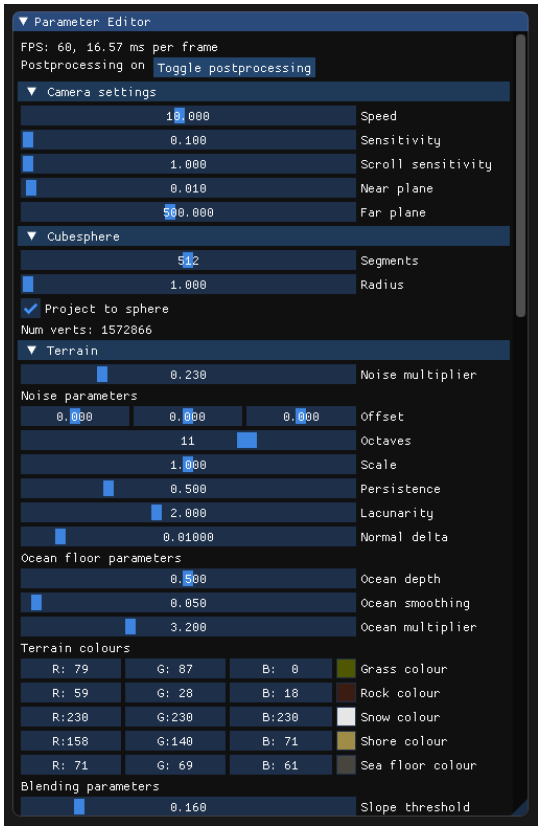

Figure 15: Parameter Editor

Dear ImGui made creating such an editor a breeze, and resulted in a lot of magic numbers abstracted away to understandable parameters.

Camera

To support user interactivity in a form other than through editing parameters, I implemented a simple first-person fly camera controllable via the keyboard and mouse. Camera parameters such as the near and far clipping planes, mouse sensitivity and fly speed can be controlled through the parameter editor as well.

Results

Below are some key screenshots and renders that showcase the key functionalities and abilities of the generator.

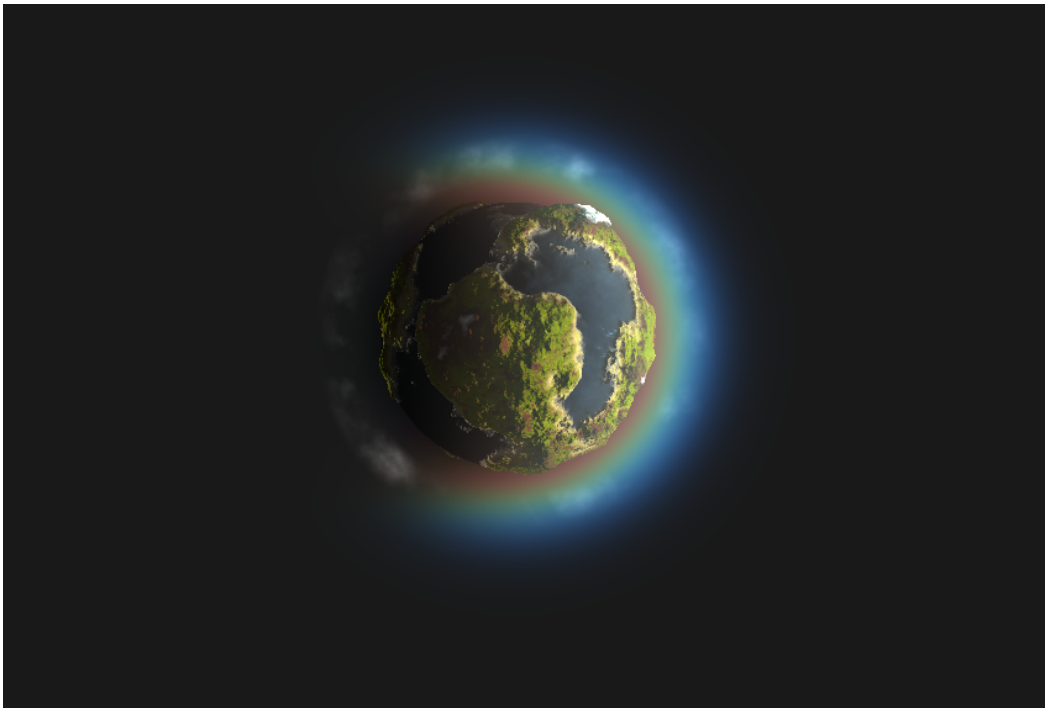

Figure 16: Space view

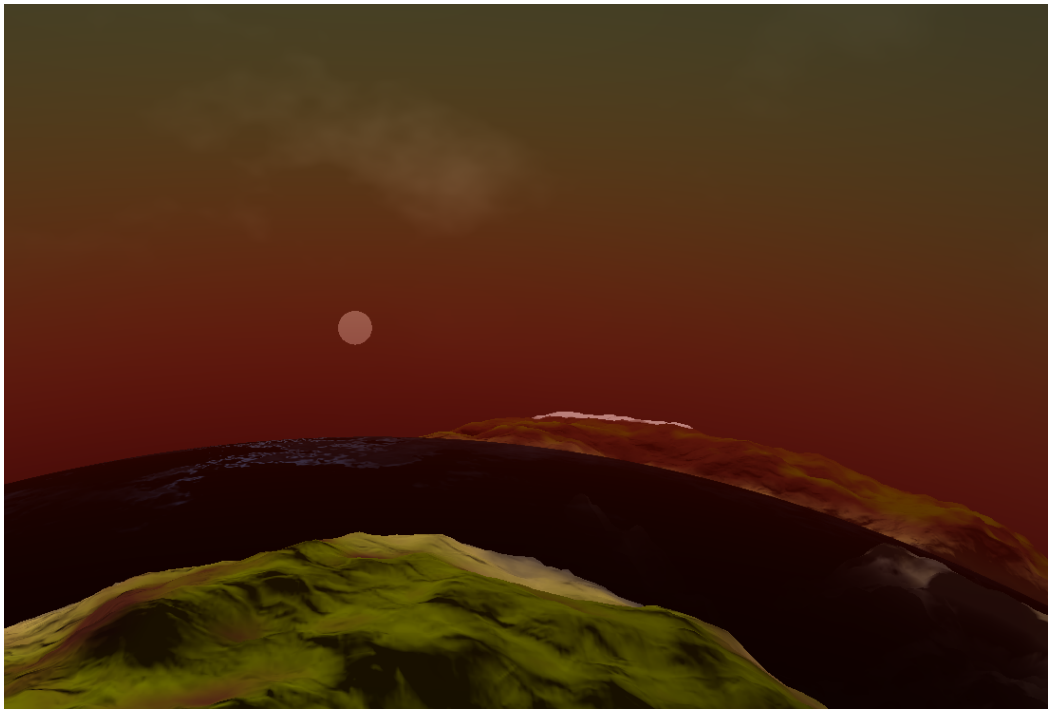

Figure 17: Red skies at sunrise

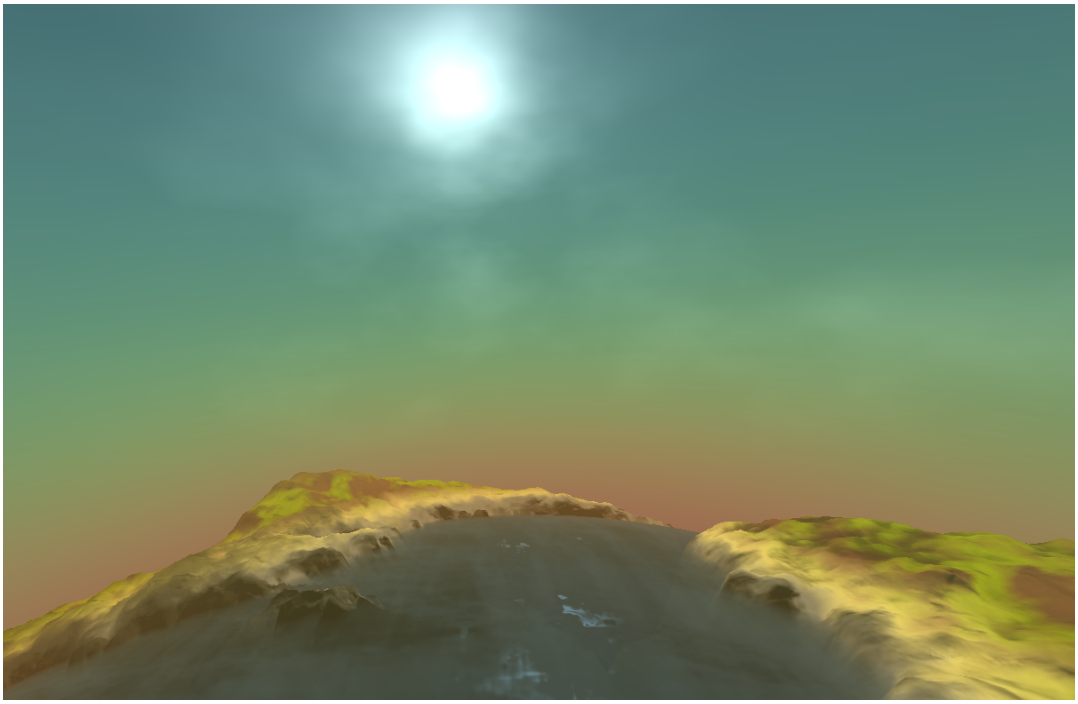

Figure 18: Blue skies and clouds

Discussion

Due to time constraints, several features were left out.

The first was the implementation of a gravity-based camera. While the current flying first-person camera works well enough for basic purposes, it lacks the ability for the user to orientate themselves with respect to the sphere. As such, the camera is really only limited to using the top pole areas of the sphere, as the ground would appear naturally at the bottom of the screen.

The second feature is related to the decision to not use pre-baked textures. This constitutes an additional optimisation step that would help improve performance. The main areas that would have benefitted from this feature would be the clouds generation (using pre-baked Worley and Perlin noise would have resulted in more realistic looking clouds), the atmospheric scattering calculations (avoiding the need to perform another expensive ray-march to calculate the optical depth), and the noise generation for the terrain. Nevertheless, the performance is relatively stable at 60 fps when seen from a zoomed out view, and only dips to around 46 fps when standing near the planet's surface and looking directly at the clouds.

One of the main advantages of my approach would be the increased performance due to computing almost everything on the GPU. Other than the vertex data, every other calculation is computed either in the vertex shader or the fragment shader.

While this is beneficial for performance, one of the issues alluded to in the Terrain section is that the vertex data cannot be easily transferred out from the vertex shader back onto the CPU. This means that collision data cannot be computed, which is relevant if the planet is meant to be used in games or the like. For this project that is only concerned with renders, however, this feature is not too important, hence why I decided to continue with computing the terrain generation in the vertex shader.

Another limitation was the choice to render the ocean as a post-processing effect. The alternative would be to render another sphere with a separate mesh data. Using a post-processing effect sacrifices some realism for performance, as a single draw call is used to render the ocean. However, the lack of manipulable vertices meant that the actual rising and falling of waves could not be realised, and thus I had to resort to using a normal map to give the illusion of moving water.

Nevertheless, even with the above limitations, the final result produces realistic-looking planets together with atmospheric scattering and voluminous clouds.

Conclusion

In conclusion, I have presented a procedural planet generator that is easily configurable and performant. It renders realistic-looking planets with atmospheric lighting and voluminous clouds.

Through this project, I have learnt a lot about the math behind various physical phenomena and how to mimic them efficiently in computer graphics. I have also learnt more about writing shader code, as well as how to render post-processing effects using framebuffers in OpenGL. I have also learnt about useful techniques such as ray-marching, fractal noise, and triplanar normal mapping, which are easily transferable to other projects as well.