Procedural Planet Generation

Algot Sandahl[1]

Abstract
This report describes how layered noise functions can be used to procedurally generate and render a miniature planet in real-time on the web. Topics covered include terrain generation, starry sky generation, and simple ways to render water, clouds, and atmosphere.

Links
Application: `https://pannacotta98.github.io/procedural-planet`
Source code: `https://github.com/pannacotta98/procedural-planet`

[1] *Media Technology student, Linköping University, Norrköping*

Contents

1. Introduction

Procedural techniques can be used to generate interesting and unique worlds, as demonstrated by games such as Minecraft and No Man's Sky. Since the release of WebGL in 2011 [1], many of the tools needed for real-time procedural generation and rendering is also available on the web. This project aims to utilize WebGL to procedurally generate a miniature planet in real-time.

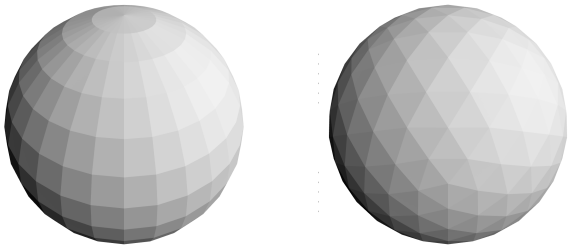
2. The Planet

The scene consists of five distinct parts, each with its own vertex and fragment shader.

2.1 Surface

The planet's surface consists of a unit sphere whose surface is displaced along the radius using layered three-dimensional noise.

The standard sphere geometry found in most software libraries is the UV sphere which is created using latitude and longitude segments. This means that the size of the polygons varies significantly over the surface of the sphere, as seen in Figure 1a. This is not suitable for this project since the planet should ideally have the same level of detail regardless of the position on the sphere. This planet uses instead the icosphere



(a) UV sphere **(b)** Icosphere

Figure 1. Sphere geometry comparison.

which is created by subdividing the faces of an icosahedron and projecting the vertices onto a sphere. As seen in Figure 1 this greatly improves the uniformity of the polygon sizes.

The altitude changes in the surface is modeled in the vertex shader by sampling multiple octaves of three-dimensional simplex noise (explained in [2]). The layered noise at the point $\mathbf{x}$ is calculated according to

$$h(\mathbf{x}) = \frac{\sum_i \left[A_i(\frac{1}{2} + \frac{1}{2}\,\text{simplex}(f_i \cdot \mathbf{x}))\right]}{\sum_i A_i} \tag{1}$$

where A_i is the amplitude, f_i is the frequency, and i denotes the noise layer. The change in amplitude and frequency for each noise layer is set by two parameters: persistence, and lacunarity, according to this pattern:

$$A_i = A_{i-1} \cdot \text{persistence}, \tag{2}$$
$$f_i = f_{i-1} \cdot \text{lacunarity}. \tag{3}$$

In addition to the normal terrain generation, a "ridge mode" can be toggled, which then replaces (1) with

$$h(\mathbf{x}) = \frac{\sum_i \left[A_i(1 - |\,\text{simplex}(f_i \cdot \mathbf{x})|)\right]}{\sum_i A_i} \tag{4}$$

where the magnitude of the the noise creates a sharp edge in the noise, and the inversion places the ridge at the top of

the mountains. However, this sharp edge can create some artifacts, so to avoid this the absolute value is approximated by a smooth function

$$\text{smoothAbs}(x) = x\frac{e^{kx}}{e^{kx}+e^{-kx}} - x\frac{e^{-kx}}{e^{kx}+e^{-kx}} \tag{5}$$

where higher k gives a result that is closer to $|x|$.

To achieve flatter fields and steeper mountains, h is mapped non-linearly according to

$$g(h) = Hh^p \tag{6}$$

before displacing the sphere surface. The total offset is set by H and the steepness change is set by p.

A challenge that arises from making these calculations in the vertex shader is that surface normals cannot be calculated using adjacent vertices. However, the normals can be approximated by sampling the height three more times per vertex, at small offsets along the surface.

First, three points in the tangent plane of the non-perturbed sphere at the point $\mathbf{x}$ are found according to

$$\mathbf{b}_1 = \mathbf{x} + \delta\frac{\mathbf{n}\times\mathbf{c}}{\|\mathbf{n}\times\mathbf{c}\|} \tag{7}$$

$$\mathbf{b}_2 = \mathbf{x} + \delta\frac{\mathbf{b}_1\times\mathbf{n}}{\|\mathbf{b}_1\times\mathbf{n}\|} \tag{8}$$

$$\mathbf{b}_3 = \mathbf{x} - \delta\frac{\mathbf{b}_1+\mathbf{b}_2}{\|\mathbf{b}_1+\mathbf{b}_2\|} \tag{9}$$

where $\mathbf{n}$ is the non-perturbed normal, $\mathbf{c}$ is an arbitrary vector, and δ is a small offset. Note that $\mathbf{b}_3$ is chosen such that the points create a triangle around the vertex. From these points the final normal is calculated as

$$\mathbf{N} = \frac{(\mathbf{s}_2-\mathbf{s}_3)\times(\mathbf{s}_1-\mathbf{s}_3)}{\|(\mathbf{s}_2-\mathbf{s}_3)\times(\mathbf{s}_1-\mathbf{s}_3)\|}, \tag{10}$$

where $\mathbf{s}_i$ are the displaced surface positions of three points around the vertex:

$$\mathbf{s}_i = \widehat{\mathbf{b}}_i\left(r + (g\circ h)(\widehat{\mathbf{b}}_i)\right), \tag{11}$$

where r is the radius of the sphere

The color of the terrain is determined by the distance to the origin, and the transitions are linearly interpolated to give a smoother appearance and to avoid aliasing. The application supports four different color bands: snow, mountains, land, and sand. The colors of these can be set freely, but the heights are fixed with respect to the water level.

2.2 Water

The planet has two different water implementations: ocean waves, and a calmer version. The calmer version consists of a bump-mapped sphere. The bump map is generated from three-dimensional psrd noise [3], and animated by rotating the gradients used in the noise generation. As mentioned in [3], this is a lot cheaper than slicing through four-dimensional noise. Furthermore, the noise implementation provides the analytical gradient $\mathbf{g}$, so the bump mapping is a simple process that is described in [4]. First the gradient is projected on the tangent plane,

$$\mathbf{g}_{\perp\mathbf{n}} = \mathbf{g} - (\mathbf{g}\cdot\mathbf{n})\mathbf{n} \tag{12}$$

where $\mathbf{n}$ is the non-perturbed normal. Then the projection is subtracted from the non-perturbed normal, and the result is normalized:

$$\mathbf{N} = \frac{\tilde{\mathbf{N}}}{\|\tilde{\mathbf{N}}\|}, \quad \tilde{\mathbf{N}} = \mathbf{n} - \mathbf{g}_{\perp\mathbf{n}}. \tag{13}$$

The ocean waves is based on the method presented in [5] which adapts trochoidal waves to spheres. In reality, there is seldom only one wave, but rather many waves that interact with each other, thus multiple waves are used to achieve the desired look. Each wave i is defined by a unit vector $\mathbf{o}_i$ pointing to the origin of the wave, a frequency ω_i, a wave speed ϕ_i, an amplitude A_i, and a steepness parameter Q_i. The position with all waves applied is then calculated as

$$\begin{aligned}\mathbf{P} = \mathbf{v}r &+ \mathbf{v}\sum_i(A_i\sin(\omega_i l_i + \phi_i t)) \\ &+ \sum_i(\mathbf{d}_i Q_i A_i\cos(\omega_i l_i + \phi_i t))\end{aligned} \tag{14}$$

where $\mathbf{v}$ is the normalized position, r is the radius of the sphere, and t is the time. Furthermore, $\mathbf{d}_i$ is the direction in which the wave travels, and l_i is a re-mapping of the latitude:

$$\mathbf{d}_i = \mathbf{v}\times((\mathbf{v}-\mathbf{o}_i)\times\mathbf{v}), \quad l_i = r\arccos(\mathbf{v}\cdot\mathbf{o}_i). \tag{15}$$

The new normals can then be calculated similarly to the position in the following manner:

$$\begin{aligned}\mathbf{N} = \mathbf{v} &- \mathbf{v}\sum_i(Q_i A_i\omega_i\sin(\omega_i l_i + \phi_i t)) \\ &- \sum_i(\mathbf{d}_i A_i\omega_i\cos(\omega_i l_i + \phi_i t)).\end{aligned} \tag{16}$$

In addition to the waves, the optical properties of water are also important to achieve the correct appearance. Water both reflects and refracts light. These properties are tricky to implement in a real-time graphics environment, and both refraction and mirror-like reflection have been left out of this project because of time constraints.

Another important property of water is the Fresnel effect which describes how the reflection coefficient R depends on the angle between the observer and the surface, and can be calculated using Schlick's approximation [6]:

$$\begin{aligned}R &= R_0 + (1-R_0)(1-\mathbf{N}\cdot\mathbf{V})^5, \\ R_0 &= \left(\frac{n_1-n_2}{n_1+n_2}\right)^2,\end{aligned} \tag{17}$$

where $\mathbf{V}$ is the light direction, $\mathbf{N}$ is the surface normal, and n_1 and n_2 are the refractive indices of the two media. The

reflection coefficient is then used to mix between a semi-transparent diffuse lighting model and a specular one. There is also the option to use standard Phong shading. Additionally, the bottom of the sea is shaded darker to simulate light falloff, with the additional benefit of making the lack of refraction less noticeable.

2.3 Clouds

The original idea was to use volumetric clouds, but because of time constraints, the clouds are constrained to the surface of a sphere. The texture is generated using three-dimensional flow noise [3]. Flow noise is similar to the layered noise used to generate the terrain, but it is combined with the rotating gradients used for the calm water, and a warp in the texture domain using the gradients of the noise. Mathematically, the flow noise used in this project is calculated as

$$h(\mathbf{x}) = \sum_i \left[A_i \,\text{psrdnoise}\left(s_i \mathbf{x} + w \sum_{k=1}^{i-1} (A_k \mathbf{g}_k),\ s_i t \right) \right] \quad (18)$$

where s_i is a scaling parameter, A_i is the amplitude, t is the (possibly scaled) time, w is a parameter that controls the amount of warp, and $\mathbf{g}_i$ is the gradient directly from the noise function for step i of the outer sum and is provided by the psrd noise implementation. The first argument to the noise function is the sampling position, and the second is the gradient rotation. The scaling doubles each step, and the amplitude works like in (2). Finally, this noise value is used to set the opacity of the cloud layer through some simple scaling and translating. Additionally, the sphere slowly rotates to achieve some additional motion.

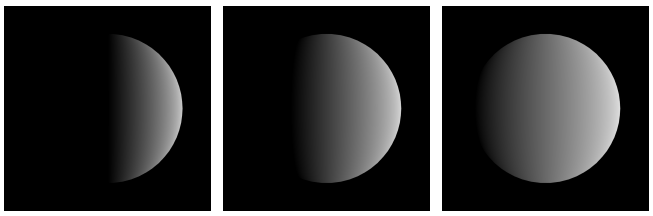
2.4 Atmosphere

To improve the realism of a planet viewed from space, the atmosphere is important. In reality, atmospheric scattering is a complicated process where different wavelengths of light are scattered differently — giving the blue glow that can be seen when viewing Earth from space. This process can be simulated, but it is complicated and resource-intensive. This project takes a simpler approach that aims to fill the same role as realistic scattering, but not necessarily by emulating it.

The atmosphere is rendered as a semi-transparent sphere slightly larger than the planetary surface. To emulate some of the effects present in real atmospheric scattering, some simple but clever tricks are used.

The opacity for each pixel is set according to a simple Fresnel term: $\alpha = (1 - \mathbf{V} \cdot \mathbf{N})^a$ where $\mathbf{V}$ is the light direction and $\mathbf{N}$ is the surface normal. This makes the atmosphere transparent when viewed straight on, and opaque when viewed from a 90°angle, and the transition in-between is set by the parameter a.

To illuminate the atmosphere, a wrapped diffuse model is used. What this means is that the light reaches further around the surface than it would with normal diffuse shading, which can be used to emulate subsurface scattering. A simple form of this technique, described in [7], is to calculate the wrapped diffuse light in the following manner:



(a) $w = 0$ **(b)** $w = 0.3$ **(c)** $w = 0.7$

Figure 2. A white sphere lit from the right using the wrapped diffuse illumination model.

$$I_d = \max\left(0,\ \frac{(\mathbf{L} \cdot \mathbf{N}) + w}{1 + w}\right) \quad (19)$$

where $\mathbf{L}$ is the light direction, and w controls how much the light is wrapped. For $w = 0$ the model reduces to traditional diffuse shading. This effect is shown in Figure 2.

2.5 Stars

The background consists of a procedurally generated star texture mapped to the inside of a large sphere. The texture is simply generated by single octave three-dimensional simplex noise run through a smoothstep function ranging only the very highest values of the noise.

3. Implementation

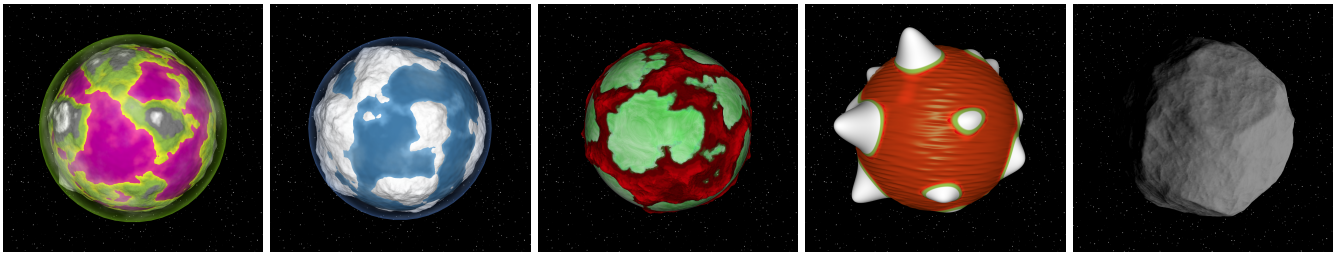
The application was implemented in TypeScript with Three.js. To achieve real-time performance, most of the generation takes place in the shaders, which are written in GLSL. To make the application more interesting, the rendered planet is accompanied by a graphical user interface where many of the parameters of the planet can be varied. The interface also contains a few presets to serve as inspiration for the user.

4. Results

The application was tested on two computers: an M1 MacBook Air with 16 GB of unified memory, and a desktop with a Ryzen 5 2600X, an RTX 2070, and 16 GB of RAM. None of the tested systems had any trouble keeping up with the application. The default preset can be seen in Figure 4, and a few other presets can be seen in Figure 3. The ridge mode can be seen in Figure 3c and 3e. Figure 5 shows the spherical trochoidal waves.

5. Discussion and Conclusion

This project successfully demonstrates how a miniature planet can be generated procedurally with WebGL using various noise functions. Nonetheless, there are things that could be improved in almost all parts of the application. Some things that would be interesting to explore are volumetric clouds and



(a) (b) (c) (d) (e)

Figure 3. Some presets.

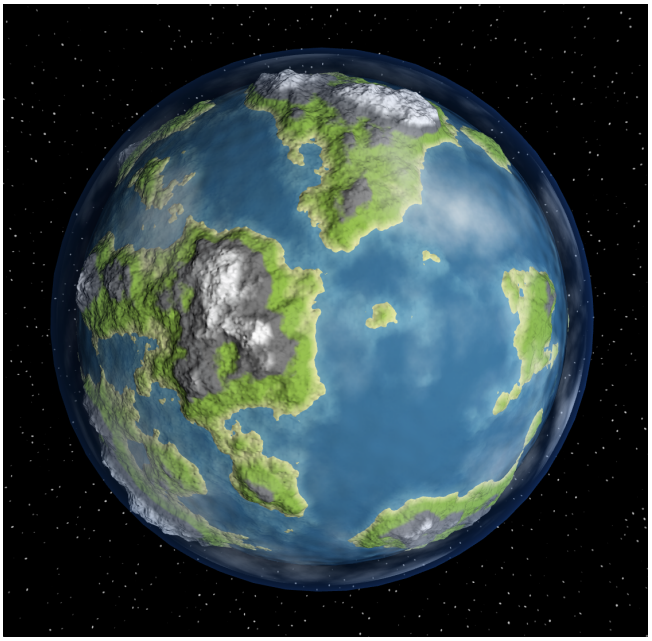


Figure 4. The default preset.

volumetric atmosphere. Furthermore, the terrain exhibits some artifacts that stem from the fact that the geometrical resolution is not high enough for the highest octaves of the noise. This could possibly be solved by rendering the higher octaves as a bump map rather than a displacement, but that would complicate the normal calculations so there was not enough time to try it. A challenge that came with the "miniature aspect" of the project was that it is hard to set the scales of different features since there are no real miniature planets to reference. Furthermore, the trochoidal waves proved very hard to configure satisfactorily. The waves seemed to always be too directional, which was a bit disturbing since the waves move with no regard to the landmasses. In conclusion, the planet is fairly convincing but could be improved still.

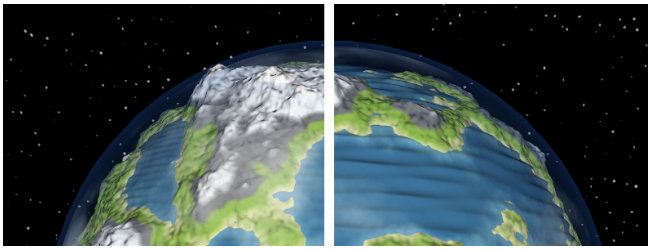


Figure 5. Spherical trochoidal waves with different sizes.

References

[1] Khronos Group, "Khronos releases final webgl 1.0 specification." Available at `https://khr.io/ba` (Accessed 2022-01-07).

[2] S. Gustavson, "Simplex noise demystified," 2005. Available at `https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`.

[3] S. Gustavson and I. McEwan, "Tiling simplex noise and flow noise in two and three dimensions," *Journal of Computer Graphics Techniques*. **Not yet published.** Pre-published version available at `https://raw.githubusercontent.com/stegu/psrdnoise/main/article/psrdnoise-article.pdf`.

[4] S. Gustavson, "Recomputing normals for displacement and bump mapping, procedural style," 2021. Available at `https://stegu.github.io/psrdnoise/3d-tutorial/bumpmapping.pdf`.

[5] F. Michelic, "Real-time rendering of procedurally generated planets," 2018. Available at `https://cescg.org/wp-content/uploads/2018/04/Michelic-Real-Time-Rendering-of-Procedurally-Generated-Planets-2.pdf`.

[6] C. Schlick, "An inexpensive BRDF model for physically-based rendering," *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, 1994. Available at `https://doi.org/10.1111/1467-8659.1330233`.

[7] S. Green, "Real-time approximations to subsurface scattering," in *GPU Gems — Programming Techniques, Tips and Tricks for Real-Time Graphics* (R. Fernando, ed.), ch. 16, Addison-Wesley, 2004. Available at `https://developer.nvidia.com/gpugems/gpugems/part-iii-materials/chapter-16-real-time-approximations-subsurface-scattering`.