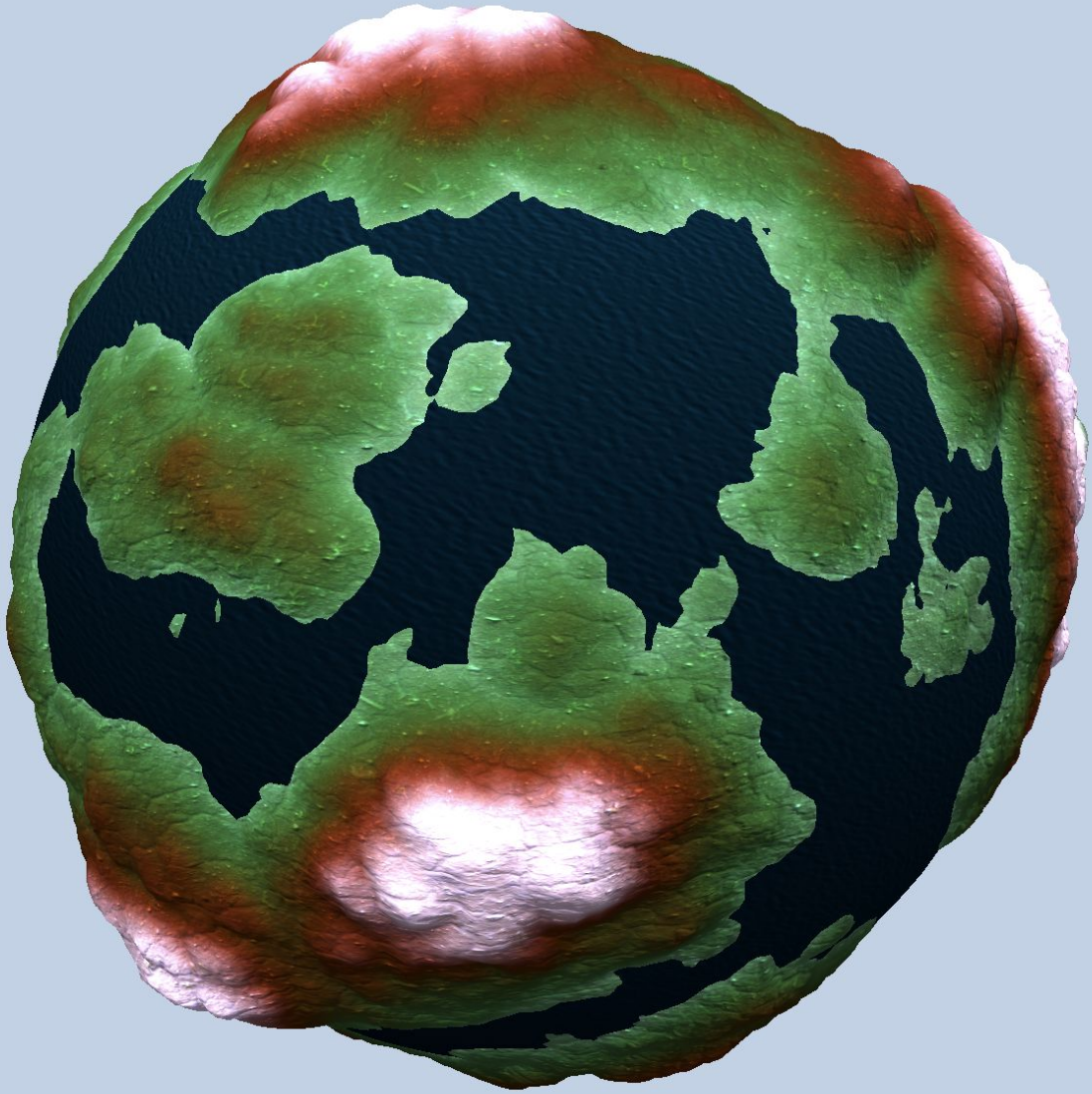
Small World

COS 426 Final Project Report / Spring 2018



Reilly Bova
Jad BEchara

Introduction

Goal

When we started our project, we had a completely different topic in mind. Instead of making a “small world” planet generator, we hoped to use machine learning to generate meshes (on a plane rather than on a sphere) that mirrored the topology of Earth. After researching papers on this topic, we made two realizations: (1) given our timeframe, it was unlikely that we would be able to achieve results as good as the paper; (2) machine learning on high-resolution planet topologies takes a really (unfeasibly) long time.

After our false start, we decided instead to learn more about established techniques for terrain and landscape topology generation. Namely, we were interested in applying the Perlin noise algorithm we implemented in the ray tracing assignment. With this new idea in mind, we came across several examples online of how this was implemented to modify the topology of a plane. Many of these results were very interesting, but they fell short of their potential as pretty art demos because they were quite limited in scope (literally to the volume of a small box). We thought that it would be cool to expand the scope of these examples either by using level-of-detail to extend the rendered topology mesh to the far-clipping plane, or to wrap the surface around into a sphere. We quickly settled on the second topic because we thought we would get more interesting results, and we believed the sphere topology idea offered more opportunities for stylization.

The final product of our project, which we have titled Small World in reference to the coffee shop that fueled us through the many sleepless nights we spent building it, is first and foremost an art demo. We believe that Small World would fit right in with the many computer graphics art demos that are popular online. In addition to this audience, we hope that our project can serve as an educational reference to students interested in learning more about topology generation and artistic planet modeling.

Previous Work

In order to build a realistic planet generator, we looked at many of the best online examples that deal with terrain generation and texture generation. Many of the previous works focus on only one of these two goals and perfect just that one goal.

Regarding terrain generation, as mentioned in the previous section, we were already aware of several examples that showcased plane-topology generation using Perlin noise. Two notable cases of this are the THREE.js example “web_gl terrain demo” and Isaac Sukin’s (Git: IceCreamYou) procedural terrain generation engine for THREE.js (THREE.Terrain). Both of these works implement the standard Perlin noise algorithm using fractal refinement, i.e. noise is generated at smaller scales and added iteratively to create a self-repeating pattern mimicking both rough and smooth terrain nicely.

When it comes to texture generation, the paper that was most useful was Ondřej Linda's "Generation of planetary models by means of fractal algorithms" which deals with many techniques for color interpolation and generation. Linda emphasizes the fact that most of the work in getting an aesthetic result is choosing the correct color palette to interpolate. Moreover, Linda proposes altitude-based linear interpolation and spline interpolation for getting nice color gradients from sea-level to mountain-top. In order to add realism and prevent every altitude from having the same color, Linda suggests using Perlin noise to either perturb the input to the 3D Perlin noise method, or to slightly perturb the noise output which is used for color interpolation, or both. Using these methods, Linda achieved results that mimic Earth.

Both approaches to terrain and texture generation described above succeed in efficiently creating realistic effects. However, both of these works are heavily skewed towards generating Earth-like planets, and the effects they produce in other cases are more synthetic and overall less successful.

Our Project

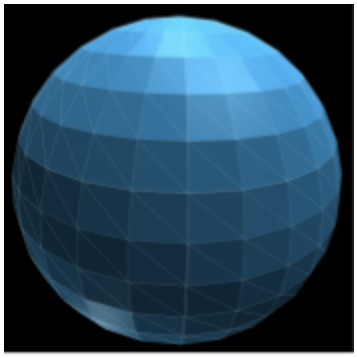
Approach

We built our project in JavaScript using the THREE.js library along with dat.gui. We settled on this approach because we were very familiar with THREE.js and dat.gui through this course's assignments, and also because we hope to one day publish this project as a web-app for all to enjoy and play around with. Reflecting on this decision, we are extremely glad that we used THREE.js. Although we did not plan to use many of its features beyond its basic rendering, lighting, and mesh tools when we started on Small World, we soon discovered that there was a lot of untapped potential in THREE.js we could use, such as PBR materials and an Ocean shader, which really helped improve the look of our project. On the downside, because THREE.js is considerably slower than C++ and other compiled graphics alternatives, we at times we had to make design sacrifices in our implementation, especially with respect to how and when we update our planet, just to make THREE.js run smoothly.

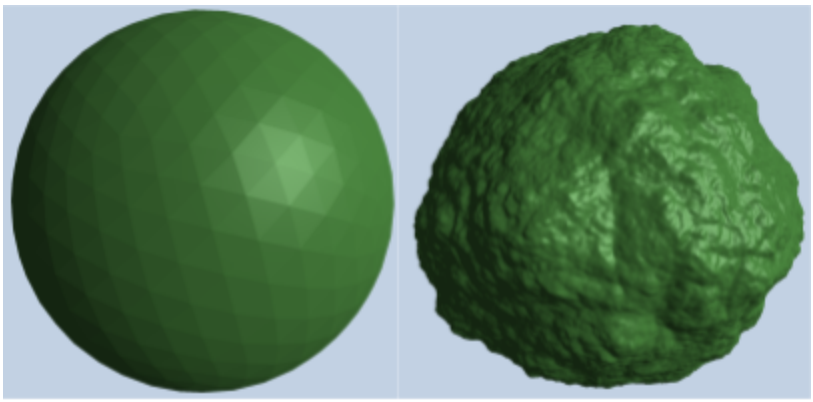
Methodology

Creating Small World was a process of repeatedly deciding on a new feature that would look cool, implementing that feature, and then refining/tuning it. Our first goal was to apply Perlin noise to offset the height of vertices on a spherical mesh. After setting up a framework for our project and digging around in the THREE.js code and documentation, we were able to accomplish this by first wrapping SphereGeometry into a new Geometry class (which we named "PlanetGeometry"), and then adding a height augmentation function in the new class.

Unfortunately, the results did not look very good because the vertices in SphereGeometry are not distributed uniformly across the surface of the sphere; instead, they are concentrated near the poles, which meant that we could not get any topology definition nearer to the equator of our spherical surface. After further research, we decided to exchange SphereGeometry for IcosahedronGeometry as the basis for PlanetGeometry. We did this because when THREE.js creates an Icosahedron mesh, it recursively applies loop subdivision to the basic Icosahedron. This results in a mesh of equal-area triangles, which means that vertices are effectively distributed uniformly across the spherical surface.

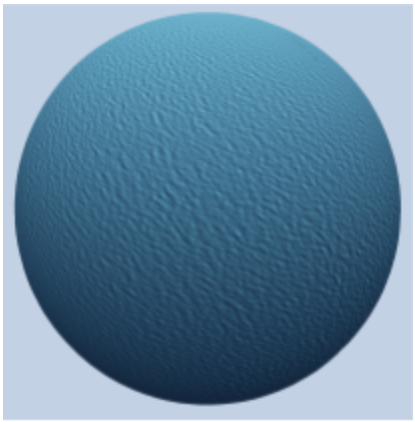


A THREE.js Sphere



A low-detail icosahedron next to a high-detail icosahedron with modified topology

When we applied Perlin noise to modify the surface topology of the Icosahedron mesh, we were immediately pleased with the results. Throughout the project, we continued to refine the parameters of the Perlin noise, even so far as to add an animation feature, but at the very least, the foundation of our project was complete after this step.



A Sphere Ocean

Next, we wanted to make our planet look more realistic. The first feature we settled on was adding a body of water. For our first attempt, we simply added a shiny blue sphere of variable height. This created nice results and offered a great improvement over our dry planet, but the result was still far from convincing. We identified two strategies to improve our water: (1) color the ocean based on depth of water, and (2) replace the blue sphere with an Ocean shader. At first, we leaned towards (1) since we figured (2) would not be feasible, but after a bit of research, we found that THREE.js offers a powerful Ocean shader for a non-spherical surface. Thus, with most of the hard work was done for us, we were able to

accomplish the second strategy by simply modifying the THREE.js Ocean such that it wrapped to a sphere, which not only looked really good, but also helped to reinforce the stylized “small world” effect that our project’s planets give off due to their exaggerated topology (just as you cannot see Mt. Everest from space, you certainly cannot see individual waves).

Once we had a believable ocean, we decided to next tackle the issue of color. In order to generate realistic colors for our mesh, we decided to apply altitude-based vertex coloring using color palettes. We approached this issue using two methods: linear interpolation (of 2 colors) and spline interpolation (of a whole color palette, using Catmull-Rom splines), where the interpolation applies to the altitude (distance from sea-level) of a point on the planet. In addition to this, we incorporated Perlin noise in our calculations to distort/perturb the contour lines (equal altitude curves) and give a natural feeling to the color distribution on the planet.

The final major feature that we added to the project was support for realistic materials. Even after the addition of biome colors, we were not satisfied with the look of our planets because the terrain looked more like polished clay than earth or rock. After a bit of research, we found that THREE.js offers full support for physically-based-rendering (PBR), a powerful shading technique that approaches photorealism when given accurate and high resolution material textures. Although we did not have PBR textures ourselves, there are wonderful free/open-source PBR libraries online that we were able to source for our project. In total, we implemented 17 materials from https://freepbr.com/, and 1 material (grass) from https://megascans.se/. We hoped to implement even more, but sadly, each material took forever to tune because we had to adjust the material parameters for each individual surface.

Spheroids of grass, wornstone, and “planet”, respectively

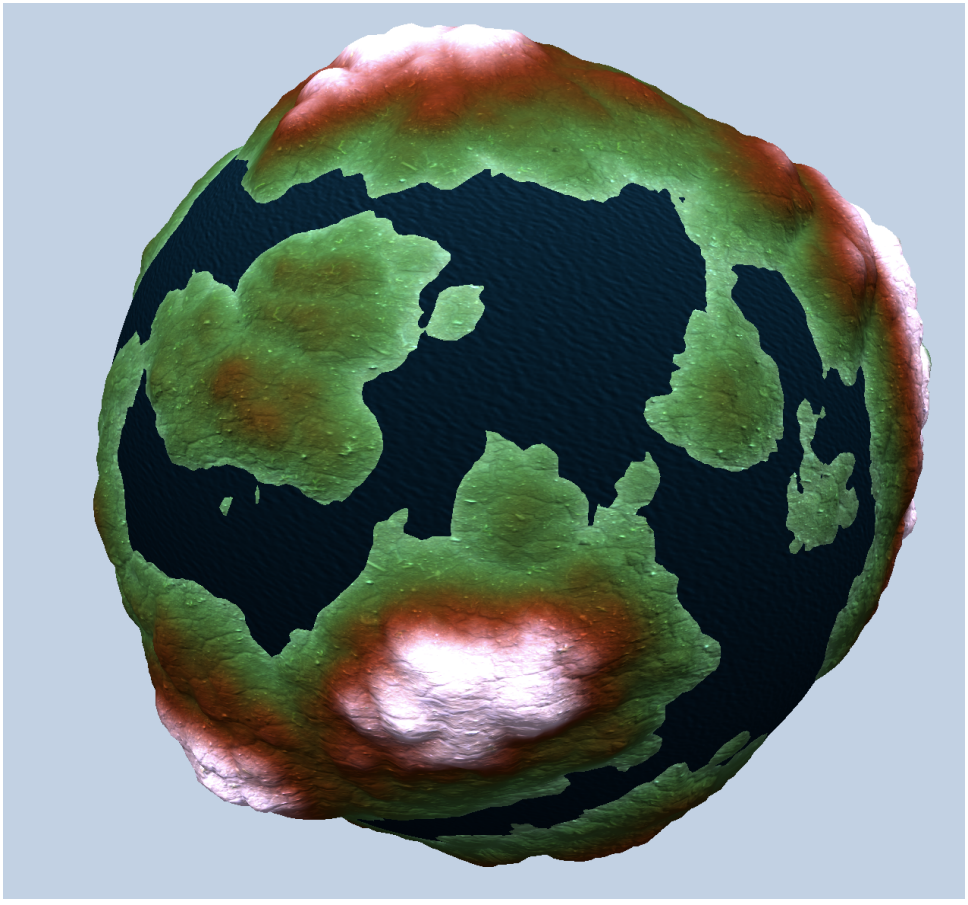
As a final few bells and whistles, we also added planet rings, sourcing a ring shader from threex.planets’s saturn, as well as a flat-shader and a toon-shader, allowing greater planet stylization. Additionally, we provided support for full recolorings of the planet and ocean, which permits for the creation of planets that look like they are straight out of science fiction.

One feature that we fell short on due to time was support for multiple terrain styles on a single planet. It was originally our hope to allow users to draw biome-maps (using colors) in a photo-editor gui, and then assigning different materials and topologies to each biome/color in the planet-maker screen. We were able to modify (namely, slim down) and link the photo editor

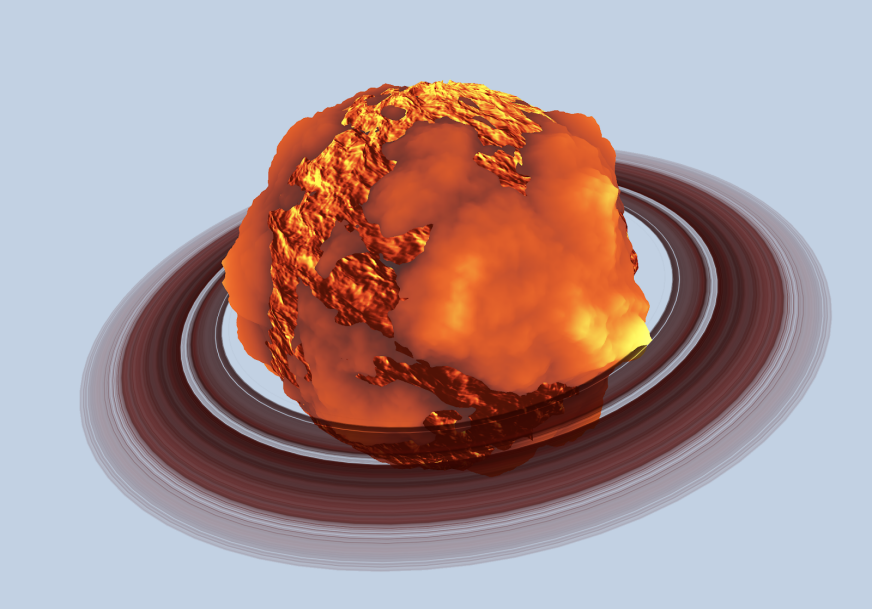
from assignment one to our assignment, and we also added a bucket-fill filter using a DFS algorithm. Despite this work, the framework of our project was such that we would have had to rewrite a lot of code in order to support multi-material meshes (which is very poorly documented in THREE.js, and potentially even a cut feature from the THREE.js library) and non-uniform topologies. Although we could not implement this feature in time for our final submission, we hope to add it in soon as we continue to refine Small World.

Results

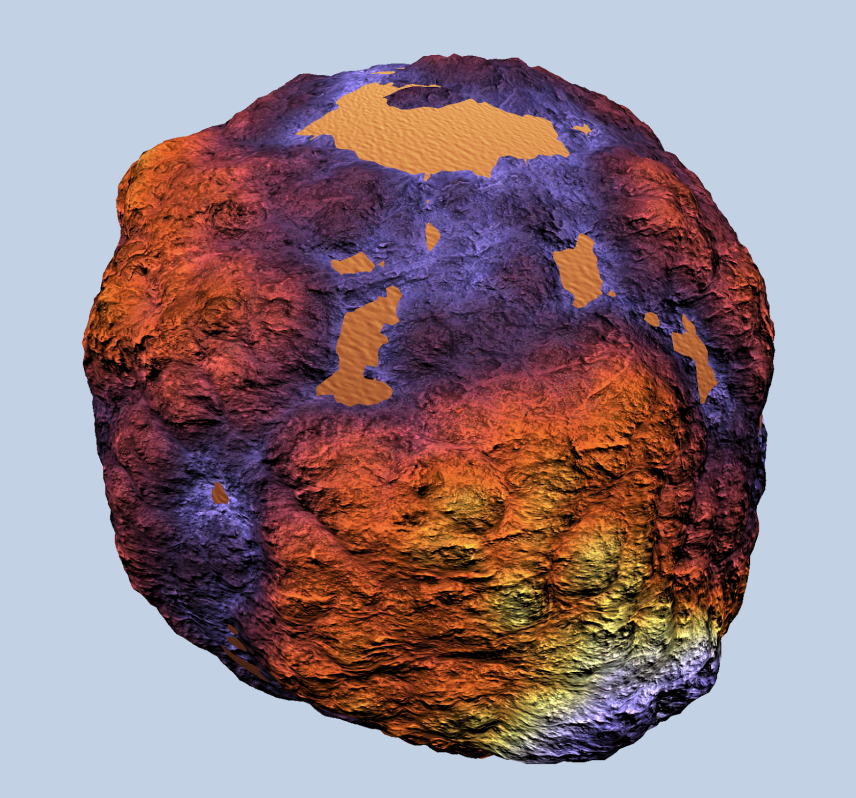
In this section, we show some of the best-looking planets we generated. We measured "success" by pure aesthetics, which corresponded to realism in some cases, and to mere beauty and awe in others. These results were achieved after tuning many of the features extensively as well as changing color palettes and materials. In the following, we will explain the intuition behind getting each of these results (and why they look good).



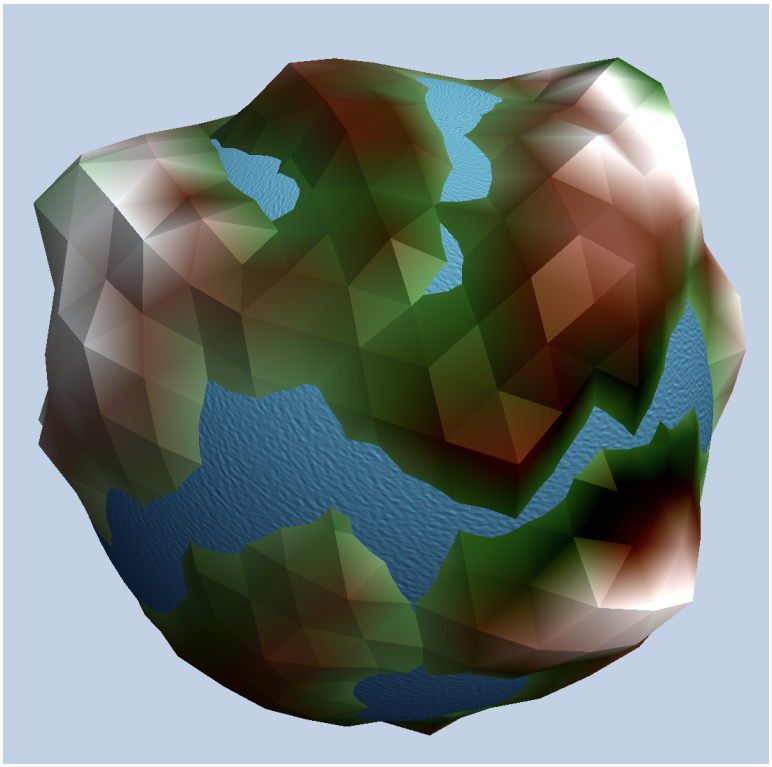
This image was generated by using spline interpolation on a green-brown-white color palette, with Perlin noise perturbation. We also mapped a dirt material to endow the planet with "veins" near the mountain-tops, thereby making it look more realistic overall.



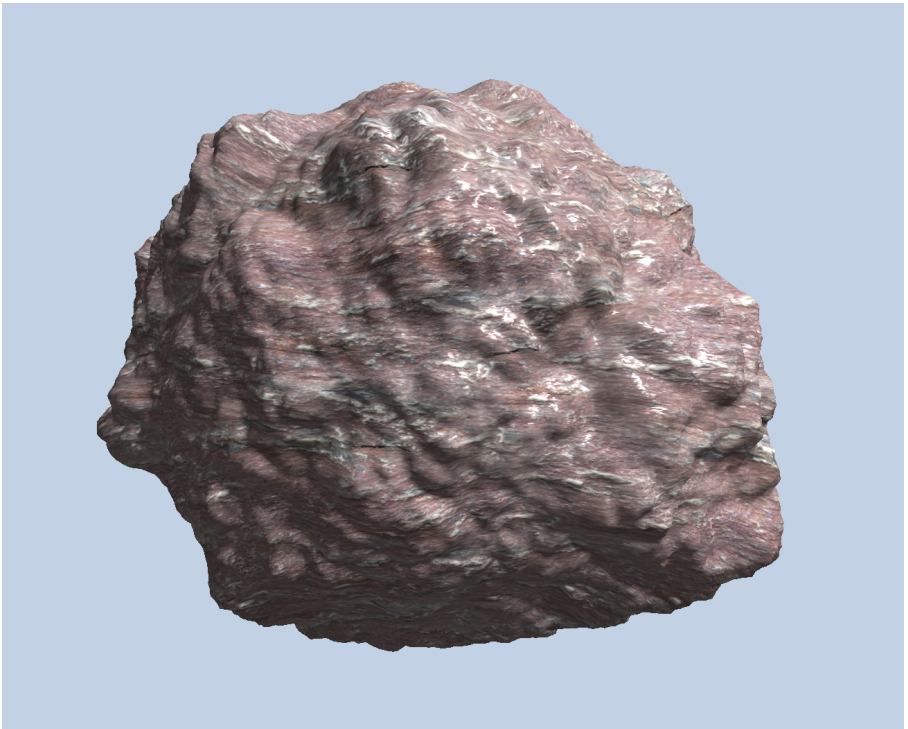
This image was generated by using spline interpolation on a red-orange-yellow color palette. The ocean's color was changed to a red-orange mix, and we increased ocean winds as well as ocean choppiness to achieve a lava effect.



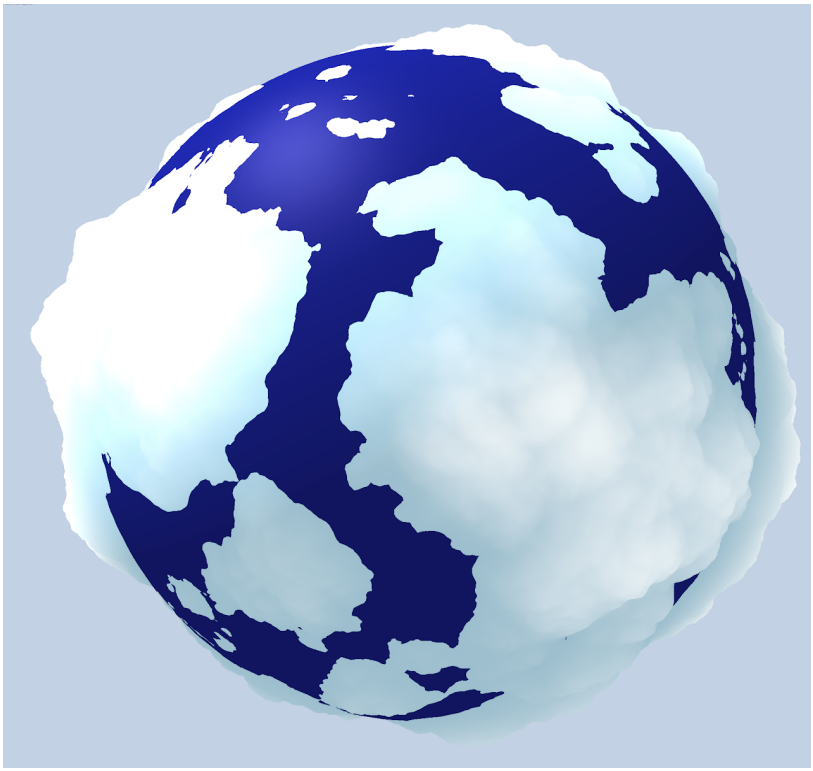
This image was generated by applying the "Nether Biome" coloring to Blackrock material. Then, the base material was colored further to give the result a purple hue. Finally, the Ocean's parameters were turned to again make it look like molten lava.



This image features a flat-shader on a low-polygon isocahedron. Although far from realistic, this gives the planet a very “retro”/stylized look.



This image was generated by applying a RedRock material map to a generated terrain, and adjusting the normal map and roughness.



This image was generated by simply using spline interpolation on a “frost palette” with tuning of Perlin noise generation.

Conclusion

Discussion

Overall, we believe that the project we undertook looks very promising: the range of possible planet-looks we observed highlights the great flexibility of our generation model as well as its efficiency. Contrary to the results of the previous works that we drew on for inspiration, our planet generator succeeds in creating aesthetic synthetic planets while also improving its ability to generate realistic planet.

There are a myriad features we envision for the future of Small World. Among those that we believe would yield the biggest improvements are:

- Multi-biome support: we want to give our generator the ability to concentrate biomes on different areas of the planet while merging them nicely (by interpolation).
- Topology editor: we want to give a user the ability to draw out a topology on a blank image, indicating the locations of mountains, valleys, and bodies of water.
- Volumetric puffy clouds: it would unarguably look exquisite.
- Saving: we want to give a user the ability to save a planet that they liked so that they can return to it later.

- First-person view: we want to build a first-person view where you can walk on the surface of the planet and explore the terrain on a different scale. This would also allow us to define new generation techniques at that scale.

Final Thoughts

In summary, through building Small World, we did not only learn both THREE.js and WebGL extensively, but we also were able to observe the interaction among all the features that we implemented. This project hence provided us with an intuition into why some combinations of materials, geometries and shaders look pleasing, while others do not, and indeed, the most intriguing part, was figuring out why. This was an extremely rewarding and fun project, and we look forward to continuing development on Small World this summer.