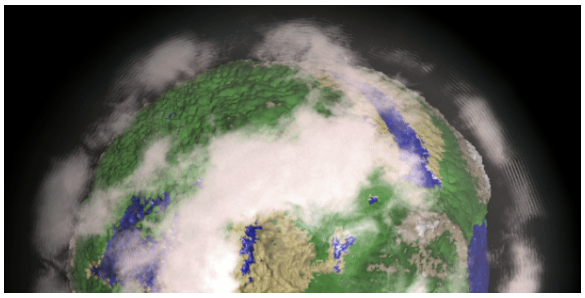

Miniature Planet Generation & Rendering

By Eric Nelson and Zack Gunther
Rensselaer Polytechnic Institute, Spring 2023

Abstract

This paper presents a procedural method for generating visually appealing miniature planets with non-physically based terrain, biomes, atmosphere, and oceans. A subdivided icosphere is used as the basis of the planet with texture maps, materials, and colors all procedurally generated via sampling Perlin noise to produce visually striking randomized planets. Oceans were rendered using low-detail spherical meshes with high-detail procedurally generated textures and normal maps. Atmospheres and clouds were rendered using concentric spherical meshes with time-dependent opacity texturing. Real-time and interactive viewings of the planet were made possible in the browser via the Quad Tree assisted LOD (level of detail) system and various shader-based optimizations.

This project was written in Javascript using WebGL for accelerated rendering. In general, the project was broken into five sections: Generating a spherical mesh, modifying triangular meshes to produce geologic-looking features, enhancing LOD with shader-generated textures, rendering oceans and atmospheres, and real-time LOD mesh generation.

Generating a Sphere Mesh

The first step in generating our miniature procedurally generated planets is to generate a spherical mesh. One common algorithm is the UV sphere where a flat 2D grid is mapped onto a sphere forming longitude and latitudinal lines [1]. This method is straightforward and simple to implement while allowing a continuous level of detail from simply adding longitudinal or lateral lines onto the sphere. UV spheres enable easy texture mapping and they are named after the “u” and “v” axis representation for texture coordinates since x, y, and z are already utilized in world coordinate spaces. One downside to UV spheres is that they suffer from uneven vertex distribution as there is a high concentration of vertices and edges near the poles as the longitudinal lines converge on a single point. Since this project will use height modulation to generate terrain this would cause undesirable artifacts as well as add complexity to LOD systems.

The second sphere generation algorithm that was investigated was the cube sphere. A cube sphere is generated by subdividing a cube using a traditional subdivision algorithm such as Loop subdivision or Catmull Clark and projecting the vertices onto a sphere. This method has vertex distribution then the UV sphere and each face of the cube can be treated as a planar mesh before projection meaning most traditional game engines and planar mesh techniques can be used on each face of the mesh [2]. Along with this, the subdividable

faces lend itself to an intuitive implementation of LOD that subdivides each face individually, storing the results with a quad tree. For these reasons the cube sphere is extremely popular in industry games such as the planet exploration game No Man's Sky. The major downside with this technique is the edges of each face of the cube cause a discontinuity in texture mapping and normal maps that is not trivial to resolve.

The third and final sphere generation algorithm we investigated was the icosphere. An icosphere is a type of geodesic polyhedron and can be generated by taking a 20-face Icosahedron and subdividing and projecting the vertices onto a sphere. A simple construction of an icosahedron is to take 3 orthogonal golden rectangles or rectangles whose side lengths add up to the golden ratio and connect their vertices to form the icosahedron [3]. From here we can utilize Loop subdivision to continually divide the icosphere producing a smooth and continuous surface without texture map discontinuities [4]. This method has the most even subdivision scheme and since the project was written from scratch we decided to base the remainder of this project on icospheres and include them in our final implementation. The downside of this decision is the loop subdivision lacks granularity with a four times increase in the number of triangles per subdivision level which is explored further in the LOD section of the paper.

An additional sphere generation technique we hope to explore in the future is through fibonacci mapping creating the most evenly distributed vertices but creating a discontinuous mesh as it needs to be mapped from a plane to a sphere [5].

Planet Terrain Generation

To generate realistic-looking terrain, a variety of different approaches were taken. All of these approaches relied upon pseudo-random or random vertex modification, with the goal of being able to create a varied or bumpy surface with large general features such as mountain ranges and valleys, and smaller, more detailed features.

The first approach explored was based on the subdivision mesh generation technique. In this algorithm, after each subdivision iteration, each vertice's position was randomly adjusted radially from the center of the sphere. This produced a mesh that appeared spikey, with no real dominant features such as mountain ranges or valleys. To account for this, the random adjustment was decreased each iteration, thus making the first random adjustments dictate the overall planet shape, while subsequent adjustments added more fine details. This algorithm was relatively efficient as it only iterated over each vertex just over once on average. As desired, this produced a mesh with varying levels of detail as well as larger, consistent features. However, it did not produce planet-like features such as mountain ranges and instead looked rather random.

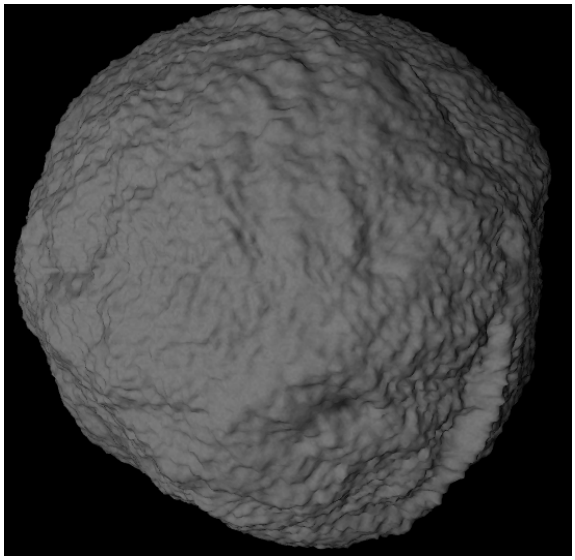

Figure 1: Gray, directionally light/shaded rendering of planet generated using Perlin noise.

The second and final approach used was based on Perlin noise, and is a common method used by many for terrain generation [6]. A 3D Perlin noise algorithm was implemented, and multiple layers of noise were added of varying resolutions to produce a continuous high-resolution 3D spatial map. Each vertices' position was then adjusted radially from the center of the sphere according to the 3D Perlin value. In order to create realistic mountain ranges and valleys, the inputs to the Perlin noise function were offset by more Perlin noise but of a slightly higher frequency. This essentially stretched the noise, stretching high and low areas into mountain ranges and valleys, creating the desired effect. In the end, many layers of Perlin noise were tested and it was found five layers of varying frequency were sufficient for generating the vertex surface features.

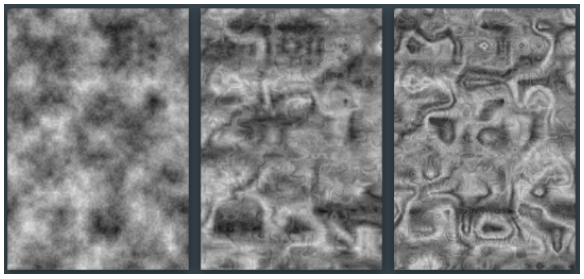

Figure 2: Five layers of grayscale Perlin Noise with zero, five, and ten times noise offset (left to right)

Biome Coloring and Materials

After generating the mesh, the next step is to color each vertex. An Earth-like color palette was selected with blue for water, yellow for sand, green for forest, brown for mountains, and white for snow. For each color, a height percentage range was chosen, and material properties were assigned. These material properties included specular reflectivity, time-dependent noise, relative noise frequency, and more which will be discussed later. To apply the colors and materials, each vertice's relative percentage radius was found (zero being the lowest vertex or vertex closest to the mesh's center, and one being the highest vertex or vertex furthest from the center), and the corresponding color and material were recorded.

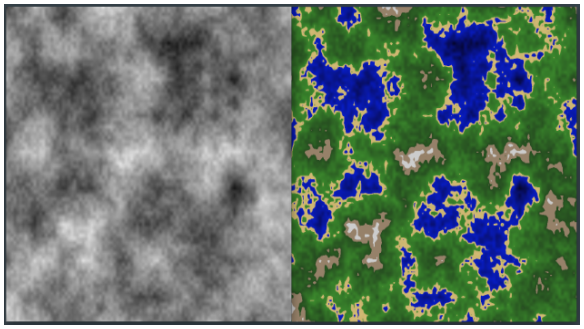

Figure 3: An Earth-like color pallet applied to a Perlin noise texture

Rendering & Shader Details

Rendering millions of triangles is computationally complex, thus reducing the triangle count in any mesh will decrease the computational cost. Unfortunately, doing so reduces the level of detail. In order to add more details than are defined in the triangle mesh, textures can be added. These textures can be used to modify vertex coloring and normal-based directional lighting but can be challenging to apply to a randomly generated mesh. To overcome this issue, it was decided the textures would be generated in the fragment shader and be mapped to each fragment depending upon the fragment position in the object's reference frame.

To do this, a 3D Perlin noise function was implemented within the fragment shader, and the vertex shader was modified to pass a varying value representing the fragment position within the object's reference frame. Then, multiple different positional noise functions were implemented to modify the fragment color and normal values depending on the surface material. For example, a specular lighting function was written to be used on reflective materials (clouds, snow, and oceans) which modified the fragment color to produce a sun-glare effect (See figure 4). Another function implemented modified the fragment normal using Perlin noise offset with more positional noise and the current time (passed in a uniform to the shader), and when ran creates a flowing effect reminiscent of ocean waves.

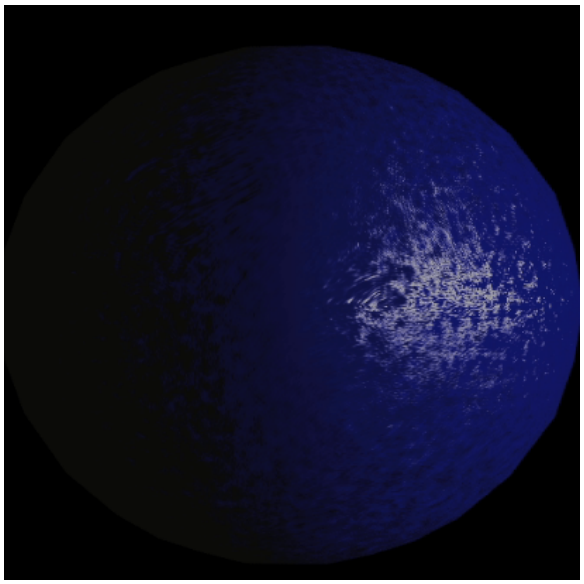

Figure 4: A blue sphere mesh rendered with directional specular lighting and the flowing ocean texture function

Altogether, these fragment shader functions, the vertex colors, and the material properties were tuned to create multiple pseudo-realistic-looking biomes, all while increasing the perceived level of detail without increasing CPU computational expense.

Ocean Generation

To create a planet model that appeared to have an ocean, a second, low-detail spherical mesh was generated. This, when rendered after the main planet mesh with a low opacity and the ocean waves shader function, created a suitable ocean. However, this method did not capture the depth of the water in the color, as deeper regions should be darker blue than shallow regions. To fix this problem, many concentric smaller meshes with low opacity were generated and rendered from the inside out. This solved the depth-color issue but was remarkably computationally expensive.

To save on computational expense, the planet mesh vertex colors were instead modified to be darker in deeper sections of the oceans.

Another issue discovered was when viewing water from a shallow angle, the water should appear darker. To fix this, the shader was modified to adjust opacity depending on the viewing angle, specifically making shallow viewing angles have a higher opacity, thus decreasing the perceived water color.

Rendering Atmosphere and Clouds

To create a realistic or artistic rendering of a planet, an atmosphere with clouds is a necessity. Commonly, rendering atmospheres is achieved by path tracing and or ray tracing algorithms, in which the distance traveled through the atmosphere is recorded and used to modify the accumulated fragment color [7]. These methods researched required a different rendering technique and were not considered for long. However, a three-pass rendering algorithm was implemented in which the first pass would render the scene to a texture, but the color of each pixel would be determined by the distance traveled to the atmosphere. The second pass would render a texture representing the distance to the back of the atmosphere or to the planet mesh. Unfortunately, this was very computationally expensive as it required rendering the entire planet mesh twice for each frame. Along with this, it was incredibly challenging to fix edge cases such as rendering from within the atmosphere, was incapable of rendering clouds, and subsequently was abandoned.

Instead, an approach that was experimented with for the ocean was applied to the atmosphere. Many white, low-opacity concentric spheres were generated and rendered from inner to outer after the planet and water meshes were rendered. As each of these meshes were of such low opacity, it was found the triangle count for each could be reduced dramatically to only 512 triangles per mesh, slightly increasing rendering performance while maintaining visual quality.

To render the clouds, a new fragment shader function was implemented. This function, using positional and time-dependent Perlin noise functions, was used to modify the opacity of each fragment in each atmospheric mesh. After fine-tuning, this function was able to create acceptable cloud formations, while not substantially increasing computational cost.

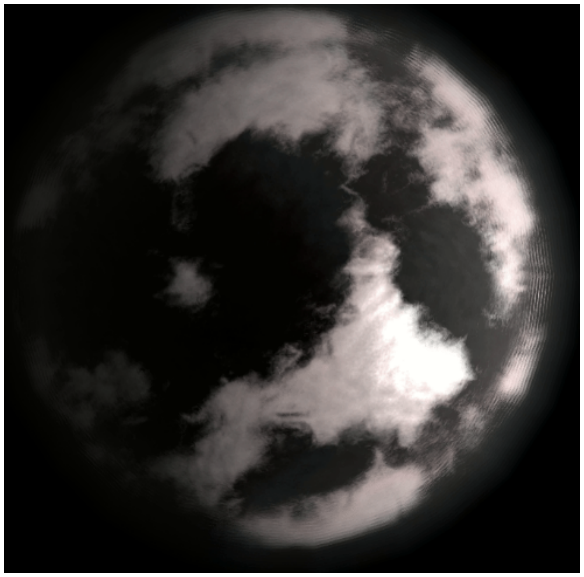

Figure 5: The atmospheric meshes rendered alone using the opacity-varying fragment shader function

Level Of Detail System

Since our project is focused on real time generation of procedural planets in the browser there are strict polygon count restrictions and performance constraints that can be eased by introducing a Level of Detail system or LOD. The LOD scheme we decided to go with was a recursive subdivision of the icosphere faces based on the location and direction of the camera. Midpoint maps are maintained to track child vertices created by two parents to reuse when neighboring triangles subdivide.

Since the Loop subdivision of the sphere results in each LOD having four times the number of triangles then the last, each division can be computationally expensive and noticeable to the end user. Preferably, this division happens out of view of the users attention (although for our demo this threshold is reduced so the LOD can be visualized) so ay distance threshold is preselected then divided by the LOD number associated with that specific triangle face squared to make the heaviest of the LOD transitions at the highest levels exponentially harder to trigger as they add an exponential number of triangles. At the highest level of subdivision a separate max depth threshold is reached preventing overloading WebGL.

With the distance threshold our subdivision scheme also considers the direction the camera is facing to ignore back facing triangles in the subdivision threshold. The orientation on the sphere of each triangle is considered to eliminate all triangles on the back half of the sphere (relative to the camera) from being considered for additional subdivision. This simple visibility detection drastically reduces the number of triangles considered for subdivision but introduces a new problem of mountains disappearing over the horizon due to their orientation of the sphere facing away from the camera but their peaks still above the horizon. These extreme mountains are common on our miniature planet and their sudden removal makes this optimization extremely noticeable.

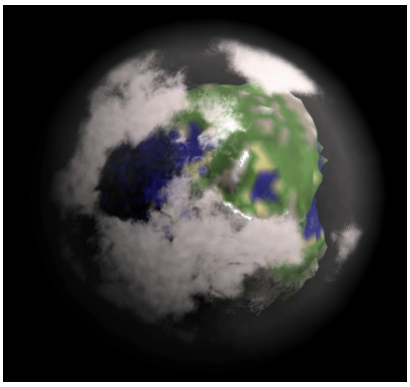

Figure 6: Miniature planet where the mountain ridges peek above the horizon.

To resolve this issue the normal of each face was considered so a triangle facing towards the camera will always be considered for sub division and an increasing threshold is utilized to make low LOD divisions easy and high LOD divisions difficult. This ensures all of the popping artifacts happen on the back face of the sphere out of view of the user while allowing for a halving of the number of triangles.

Level Of Detail: Quad Tree

The recursive nature of the LOD system lends itself well to an quad tree based caching method for storing computed LOD. This drastically reduces frame stuttering associated with terrain generation pauses. The quad tree we decided to implement stores each triangle as a node and

all of its subdivided triangles as children to allow access to individual triangles in logarithmic time. This quadtree is wrapped in the SurfaceTree class where the 20 faces of the icospheres become the roots for this quadtree mesh. Storing the terrain in a quad tree allows for the colors, materials, and vertices to all be stored in large buffers while only the indices of the triangles are searched for the triangles to export for each frame's LOD. Once the quad tree is recursively descended to collect the indices for each frame only the needed vertices, colors, and materials can be sampled and sent to the rendering pipeline to reduce load on the shaders. After the introduction of the quad tree the LOD system was operating in real time with a majority of the performance hit coming from loading the object buffers with different data each frame causing frequent garbage collection. This could be avoided in the future by restructuring the buffer management.

Level Of Detail: Stitching Holes

LOD implementations frequently cause mesh discontinuity issues where the edges between different LOD have vertices along edges that aren't shared [8]. For our project, the Perlin based height modulation may place a midpoint below or above the edge corresponding to its parents causing a noticeable hole in the mesh.

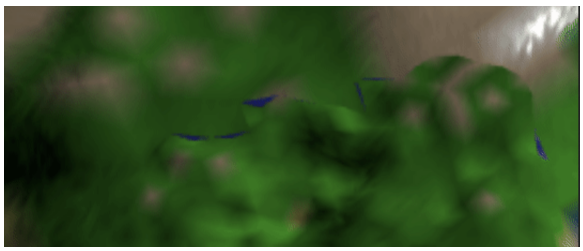

Figure 7: Holes (in blue) formed between the border of a high LOD (bottom) and a low LOD (top)

To resolve this the mid point map created earlier to ensure midpoint vertices were shared among neighboring triangles was modified to record the number of accesses. If an edge was accessed in the descent of the quad tree an even number of times it must be shared by two triangles in the final mesh and there is no discontinuity. If the edge was only accessed an odd number of times it means the edge is at the border of two different LOD. Since only the even or oddness is important this is represented by a boolean flag indicating whether the edge is a border or not.

Once the border edges have been identified the midpoints of the higher LOD mesh are simply moved to be along its parents edge thus making the discontinuity in the mesh invisible [8]. With the holes removed the LOD border is less distracting as it progresses along the terrain.

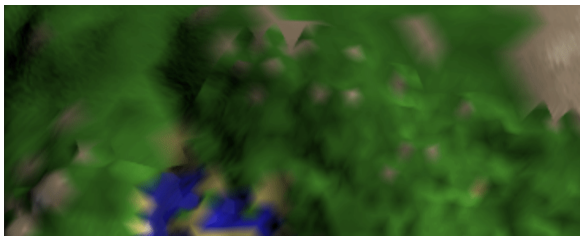

Figure 8: Border of a high LOD (bottom) and a low LOD (top) with the stitching algorithm applied

Results and Conclusion

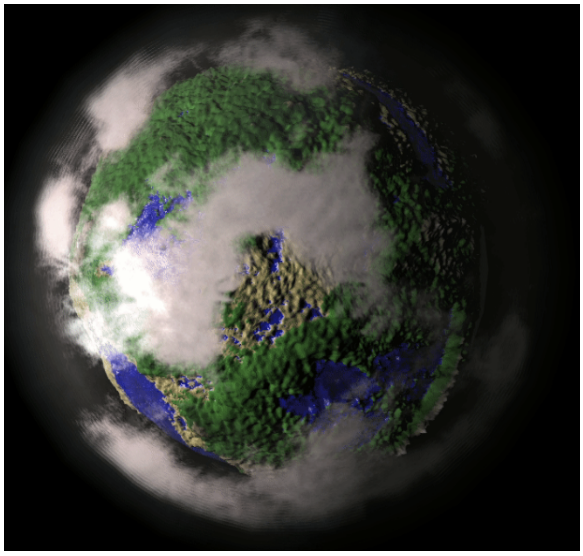

Figure 9: Final rendering of the generated planet with oceans and atmosphere

Overall, a planet generator was developed capable of creating pseudo-realistic miniature planets. A system for rendering planet meshes was developed and is capable of rendering a variety of planets with different colors and textures through the color and material vertex attributes. Along with this, a method for rendering acceptable clouds and oceans was developed, all of which do not require path tracing or raytracing. In the end, this generator and renderer can run in real-time in a web browser at acceptable framerates (approximately 55 fps when running on an Intel i7 9th gen CPU with a GTX 1650 graphics card).

However, as the time constraints for this project were rather severe, there are many improvements that could be made to this project. Firstly, improvements could be made to the planet mesh generation by adding asynchronous abilities to reduce page load times. Along with this, an addition of the sqrt(3) subdivision paper would help reduce the exponential increase of triangles in the LOD from 4 times to 3 times with each subdivision [9]. More could also be done to tune the LOD mesh generation to reduce visible differences between separate LOD sections. Finally, more features could be added to the shader to allow for a multitude of different features. As an example, cloud color modifiers could be added to enable clouds to be different colors than the atmosphere.

Project Roles

Eric Nelson:

- Spherical Mesh Generation
- Level Of Detail System
- Quad Tree
- Integration of the non-LOD and LOD mesh generations
- Planet.js implementation

Zack Gunther:

- Planet mesh vertice modification
- Vertex coloring and material assignment
- WebGL rendering pipeline with lighting
- Shader texture generation
- Ocean generation and shader effects
- Atmosphere and cloud generation and shader effects

References & Resources

[1] Mahdavi-Amiri, Alderson, and Samavati, “A survey of Digital Earth representation and visualization,” PRISM Home, 07-Apr-2015. Available: https://prism.ucalgary.ca/handle/1880/50407

[2] Zucker & Higashi, “Cube-to-sphere projections for procedural texturing and beyond,” The Journal Of Computer

Graphics and Techniques, vol.7, no.2, 1-22, 2018

[3]
Josephine Sheng, “Take and Make: Icosahedron from Golden Rectangles, ” MathHappens Blog, May 6, 2021. Available: https://www.mathhappens.org/take-and-make-icosahedron-from-golden-rectangles/#:~:text=The%20object%20in%20the%20photo,(1%3A1.618033%E2%80%A6).

[4] Charles T. Loop. Smooth subdivision surfaces based on triangles. Master’s thesis, Department of Mathematics, University of Utah, August 1987.

[5] Keinert, Innmann, Sanger, and Stamminger, “Spherical fibonacci mapping,” AMC Transactions on Graphics,” vol. 34, no. 193, 1-6, Nov, 2015

[6] Rose & Bakaoukas, “Algorithms and Approaches for Procedural Terrain Generation - A Brief Review of Current Techniques,” Research Gate, vol. 10, no. 1109, Sep 2016

[7] Sébastien Hillaire, “A Scalable and Production Ready Sky and Atmosphere Rendering Technique,” Eurographics, vol. 39, no. 4, 2020.

[8] Nick Gildea, “Fixing a seams bug,” Nick’s Voxel Blox, July 26, 2015. Available: https://ngildea.blogspot.com/2015/07/fixing-seams-bug.html

[9]
Leif Kobbelt, “Sqrt(3)-Subdivision,” Max-Planck Institute. Available: https://www.graphics.rwth-aachen.de/media/papers/sqrt31.pdf