Hydraulic Erosion on a Planet Like Surface
Senior Research

Maui Kelley

November 2020

1 Introduction

When playing open-world games you are given a world to explore. Most of the time this world is a flat piece of terrain that is either handcrafted to the experience like in the game "The Legend of Zelda: Breath of the Wild"[7], or it is procedurally generated like in the game "Minecraft"[6]. Both of these approaches have their advantages and disadvantages. The major advantage procedural worlds bring to the table is its endless content however, its major disadvantage is the uniqueness of said content. This is the worst flaw procedural generation can have since its selling point is the amount of content. I believe this can be fixed by upgrading the way we add depth to our procedurally generated worlds.

The current way of thinking is to add more prefabs, handcrafted structures that can be randomly placed throughout your world, to the terrain generation. To me, this seems like a band-aid to a bigger problem. If the terrain can't stand on its own as fun to explore, then there is an inherent flaw that needs to be addressed. The standard for generating terrain quickly is layering different levels of Perlin noise to create realistic height maps for their terrain, while this is a great starting point it lacks any depth.

Others have tried to address this problem by adding erosion simulations to the terrain to make each little hill feel unique however, if you erode every hill the same amount then every hill will look similar. Another way people have tried to add variety to their terrain is by adding biomes to their world, and like the other things that have been tried this has its flaws. For example, a snowy biome like a tundra can be right next to a dessert, or a lush forest. I set out to take all of these ideas and incorporate them into a sphere. I got to the point where I have a sphere generated with Perlin noise to create a starting point and a weather and erosion simulation to give depth to the mesh.

2 Related Work

The inspiration for this project came from a YouTuber with the username Sebation Lague. He uploads small coding projects that he works on in his free time. The video that inspired me took a look into terrain generation and was called "Coding Adventure: Hydraulic Erosion"[4]. In the video, he talks about how he implemented the erosion simulation described in Bayer's paper[1] in the Unity game engine. While I implemented the same simulation for my erosion step, I took it further by applying it to a sphere instead of a plane, and I also added a layer of complexity with a weather system.

Sebastian also has a video series[5] where he creates a procedural planet generator in the Unity engine however, he doesn't take the idea as far as I did. He creates planets with a Perlin noise height map and uses latitude to determine biomes, but he doesn't have any weather or erosion applied to the mesh. I took the idea further by having the weather and erosion systems I mentioned before.

The erosion algorithm that I used is described in Bayer's paper[1]. It uses the

idea that small particles will be picked up by a drop of water moving downhill with momentum and when that drop stops that's where the particles it's holding get dropped off adjusting the heights on the heightmap. I expanded on this by only eroding an area by how much weather it gets in a weather simulation run on the terrain of the planet. I used NOAA's Weather Systems & Patterns article[8] to familiarize myself with how the earth's weather systems work on a basic level. Using the basic rules described in this article I created a somewhat realistic weather simulation, by having air masses move rain clouds across my planet.

3 Method

The goal of this project is to create a planet generation system that can be used for creating unique game worlds. In the time span of this semester, I got to the point where the terrain is generated on a sphere and an erosion simulation can be run on the surface of the planet. The simulation can either be run on the entirety of the planet's surface or it can restrict the erosion to a weather system. For the future, I would like to implement biomes based on the weather system and latitude. After I have that I can populate the surface with biome specific prefabs, like cacti in desserts and trees in forests. I would also like to implement a way to export the meshes generated to an obj or fbx file.

3.1 Icosphere

The First thing I needed to do was make a sphere mesh that I could morph into the planet. The most important thing though when it comes to making this sphere is making sure that the vertices are evenly spaced across its surface. To start with I used the Unity Engine to create my simulation and I used Bayer's paper[1] and Sebation's[4] implementation to guide me when implementing the hydraulic erosion. The first thing I had to do was generate a sphere. I had several options here, a UV sphere, a cube with vertices spaced from the center evenly, and an Icosphere. A UV sphere is a sphere made up of latitude and longitudinal lines however this causes the poles to have a high density of vertices meaning the poles would have more detail than the equator. A cube is better but it causes bunching at the areas around the corners of the cube. Finally, there's the Icosphere, this is our best option since all of the faces are the same size and the vertices don't get bunched up. I've shown the edges of these three types of spheres in figure 1. The normal algorithm to generate an Icoshpere is to start with an Icosaherdon and split each face into four equal-sized faces. The number of times you split the faces determines how detailed your sphere becomes as you can see in figure 2. The problem with this is the number of vertices increases exponentially and the time it takes to do a new split also grows exponentially. If on the first split you split each face into more than just four faces, you can control the number of vertices with greater precision. So that is what I did. I also keep a map of what vertices have been created so that checking if a vertex

is a duplicate approaches constant time, O(1). With that, I now have a way to create a sphere with evenly spaced vertices. This algorithm also allows me to control the number of vertices with a higher level of precision than other Icosphere algorithms.

Figure 1: Visual for edges of a cube sphere, UV sphere, and Icosphere.

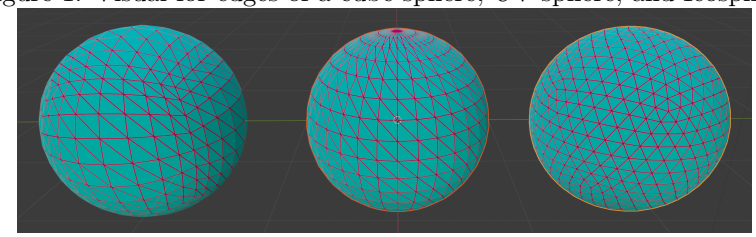
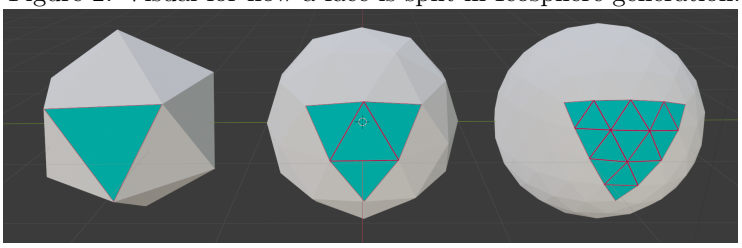


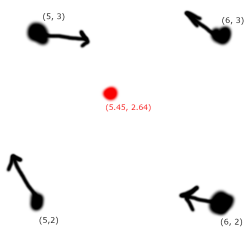
Figure 2: Visual for how a face is split in Icosphere generation.



3.2 Terrain

After I have the sphere generated I need to create a base terrain. This is necessary because the erosion simulation is designed to move sediment downhill. If there isn't a hill, then there isn't clear direction sediment should move and terrain wouldn't form. To do this I used three dimensional Perlin noise. I used a site called Ranja Tutorials[2] to learn how to implement Perlin noise. To explain how 3D Perlin noise works I need to explain how normal Perlin noise works. First I need a way to get a "random" vector that is always the same given the same input vector. To do this in code I do several mathematical functions in conjunction with another set of constant vectors to give me a pseudo-random vector. Now that I have a way to turn coordinates into a random vector, I can get a random vector associated with each of the integer coordinates surrounding the coordinate we are looking at. So to

Figure 3: The random vectors of the integer points surrounding the example point.



explain this with an example, if I have the coordinate (5.45, 2.64) I can get the random vectors associated with (5, 2), (5, 3), (6, 2), and (6, 3). These are the four integer coordinates surrounding (5.45, 2.64) see figure 3. Then for each of those vectors we created, we can get the difference between it and the vector pointing at our base coordinate using the dot product. This will tell us how much the coordinate is in the direction of the vector of each point. Knowing this we can linearly interpolate between all the points to get how much each direction is pointing at the coordinate giving us a noise value for the coordinate that smoothly transitions between points. This is how you generate Perlin noise. To do this in three dimensions, you just generate a vector for the eight integer points surrounding the coordinate. I then layer four calculations of Perlin noise with different cell sizes and weights giving me a single noise value for each coordinate. This value is between zero and one so if I multiply it by an altitude I can set that to the magnitude of each vertex to create the terrain see figure 4.

Figure 4: Perlin noise applied to a sphere.

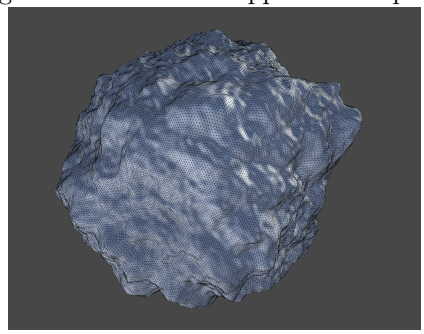
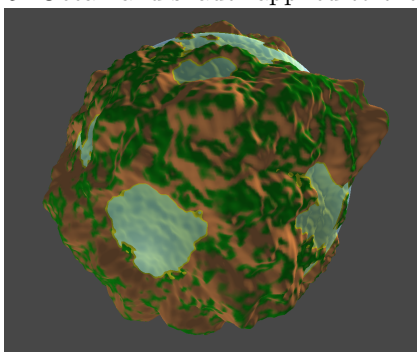


Figure 5: Ocean and shader applied to the planet.



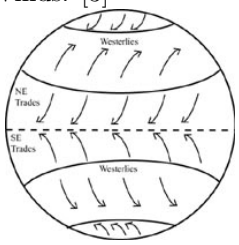
Now that I have terrain I can set an ocean level in which I create a sphere with a radius of ocean level and give it a semi-transparent blue material. This creates a visual representation of water. Next, I need to set the normals of the mesh so that we can determine things like the downhill direction and steepness

of a cliff. Normally the way you would do this is to calculate the cross product of the two vectors between a face's vertices in a clockwise direction. However, since Unity's coordinate system has a flipped z-axis you must calculate it in a counter-clockwise direction. Using the calculated normals I then apply a shader to the planet that colors the mesh a set color below ocean level, a set color for terrain of some level of steepness, and another set color for the rest of the mesh, the results are seen in figure 5. At this point, I have made what looks like a planet. However, the bumps and hills don't look like mountains and the terrain itself is very bland.

3.3 Weather

So at this point, I have a sphere with the terrain and an ocean. The next thing I implemented was a weather system. The weather system is important because it can create areas that receive a lot of erosion and areas that receive little erosion. This is more realistic than everywhere on the mesh getting the same amount of erosion. The way this works is by having six zones on the surface of the planet separated by latitude. These six zones were all given a direction of wind flow that is somewhat the same as earth's, see figure 6. So when I start the simulation with weather turned on I made it so every frame there is a chance that a cloud spawns above a vertex that is below sea level. clouds have a set amount of frames that they will exist and a velocity. Their velocity gets updated by the wind of their current zone every frame. This is how the weather system works. A cloud spawns above the ocean and moves across the surface of the planet based on the wind zones. Finally, I have everything I need to implement erosion simulation.

Figure 6: Earth's Trade Winds. [3]



3.4 Erosion

The erosion simulation is the key to making the planet look more realistic. The erosion simulation's goal is to smooth out pits and bumps, as well as carve erosion lines in the sides of mountains. This creates a realistic and interesting looking terrain. At a basic level, the way it works is by simulating a droplet following the path of least resistance down the terrain eroding sediment in its path and depositing it else ware. Doing this relative to the surface of a sphere is complicated. The steps I needed to take were to determine a spawn location, calculate the direction of least resistance, get the next position of the droplet, calculate the difference in altitude of the previous position, calculate the sediment capacity for the droplet, check if we should erode or deposit sediment and do that, and finally update the droplets position and speed. We do this a set amount of times per droplet. I have two ways of determining a spawn location. If the weather system is turned on a droplet is spawned at a random point below

every cloud. If the weather system is turned off I spawn a set number of droplets at random points on the surface of the planet. Then to calculate the path of a droplet of least resistance. To do this I first get the face that the droplet is on. Then I create a quaternion for the rotation of the unit z-axis to the normal of the face and rotate the vector from the droplet's position to the origin by that quaternion. If I set the z value of that new vector to 0 and rotate it by the inverse of the quaternion I get a vector pointing in the direction of downhill. I can then use the droplet's current direction of movement and the new direction to calculate a direction that is pointing slightly more downhill. This is the path of least resistance. I then need to get the next position of the droplet. Instead of moving the droplet by the speed in the direction of least resistance, I only want to move it by the distance between vertices so that a droplet hits every vertex in its path. So I do that and get the next position. Using this I can calculate the difference in altitude which tells me the max amount I can erode the current position. If I eroded more than the difference in altitude it would create pits from the droplet going back and forth between two vertices. I then get the maximum amount of sediment a droplet can hold, which is higher when moving fast. Then if the sediment in a droplet is more than the max sediment or the droplet is moving uphill I deposit either the difference in altitude if the droplet has that much sediment or the amount of sediment it does have. Otherwise, I erode a set fraction of the amount of capacity left, but no more than the difference in altitude. I finally calculate the droplet's new speed and evaporate a fraction of its water. This happens a set number of times until I remove the droplet. After running the simulation for a while more interesting terrain will form see figures 7 and 8. At this point the simulation is complete. The steps I took were to generate a sphere, apply Perlin noise to create terrain, create a weather simulation that determines what areas get the most rain, and finally create the erosion simulation that morphs the terrain.

Figure 7: End of simulation with weather turned off.

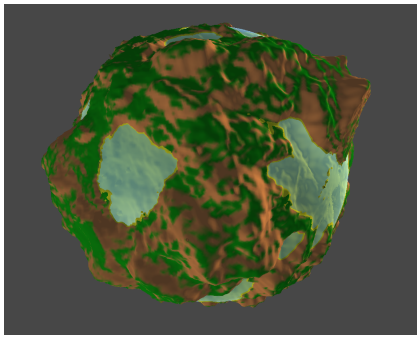
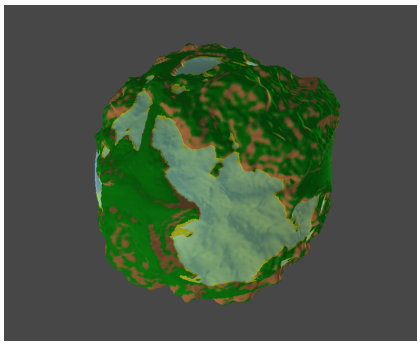


Figure 8: End of simulation with weather turned on.



4 Results

I found that running the simulation with five thousand iterations for weather turned on gives the best results. However, the terrain generated when the weather is turned off is far superior, ten-thousand iterations with ten droplets per frame gave the best results. I believe this is because the weather system is so bare bones. Because there is no other factor to the movement of a cloud than the set wind directions, only the equator and poles get much rain. I think this can be improved by having clouds move around mountains and having wind paths that are a little more complicated than the general patterns I used. Although the transitional areas between areas that get heavy rain and little rain do look good and I wasn't able to get similar results in the simulation without weather.

5 Conclusion

The impact of the weather system on the erosion simulation wasn't as groundbreaking as I thought it would be, however it did show promise. I think that if I improve the weather system to be a little more complicated I can get the results I was hoping for. The next steps I would take with this project are to improve the weather system, create biomes that affect visuals and erosion behavior. Then finally I would populate the surface with prefabs like trees and foliage. At which point I would say the project could be used in the production of games that need planet shaped worlds.

References

[1] Hans Theobald Beyer. *Implementation of a Method for Hydraulic Erosion.* URL: `https://www.firespark.de/resources/downloads/implementation%20of%20a%20methode%20for%20hydraulic%20erosion.pdf`. (accessed: 03.25.2020).

[2] Ronja Bohringer. *Ronja's Shader Tutorials: Perlin Noise*. URL: https://www.ronja-tutorials.com/2018/09/15/perlin-noise.html#perlin-noise.

[3] University of California. *Trade Winds and the Hadley Cell*. URL: http://earthguide.ucsd.edu/virtualmuseum/climatechange1/08_1.shtml.

[4] Sebation Lague. *Coding Adventure: Hydraulic Erosion*. URL: https://www.youtube.com/watch?v=eaXk97ujbPQ.

[5] Sebation Lague. *Procedural Planet Generation*. URL: https://www.youtube.com/playlist?list=PLFt_AvWsXl0cONs3T0By4puYy6GM22ko8.

[6] Mojang. *Minecraft*. [Windows Executable]. 2009.

[7] Nintendo. *The Legend of Zelda: Breath of the Wild*. [Switch Cartridge]. 2017.

[8] National Oceanic and Atmospheric Administration. *Weather Systems & Patterns*. URL: https://www.noaa.gov/education/resource-collections/weather-atmosphere-education-resources/weather-systems-patterns.