Robbie Freeman

Tommy Martinson

5/12/20

Dynamic Planet Rendering with Noise Generation

Abstract

We set out to create a model that can represent a planet and render it in more detail as the user zooms in more. In order to do this, we attempt to implement an algorithm that can present to the user more detailed generated views of a spherical mesh as well as its surrounding texture.

Introduction

Our goal in undertaking this project was to procedurally generate detail on a planet-like sphere. Our vision is similar to a generalized Google Earth, where instead of Earth the planet could have any planet-like mesh. This can be interesting for a few different applications, like examining the effects of noise as more noise is generated from previous levels of noise, creating realistic worlds and terrains.

Approach

First, we set out to make a prototype in the form of a plane. We wanted to set up the environment so that as the user zoomed into the plane, we would subdivide the triangles on screen. Once we had the plane working, we were going to generalize our solution for a spherical mesh. Of course,

this beginning part would just focus on geometry, but the ultimate goal was to introduce detail enhancement for both geometry and the color of the texture. In order to manage zooming in and out, we planned to have different thresholds of distances, where the view of the planet at one threshold is blended with the view of the planet at the next closest if the user is zooming in, or the one further if zooming out. Eventually, the planet upon being zoomed in enough would instead be rendered as a plane, where the detail is based on the details of the planet at the least level of detail. We have decided to use the starter code and to use Three.js because we are most familiar with this WebGL.

Features

- Choose Texture - allows the user to choose the texture for mapping onto the sphere
- Different types of noise generation
    - Linear interpolation - each level of generation just linearly interpolates the previous layer's geometric and material information using reverse mapping.
    - Perlin noise - at each new level, perform linterp to get a set of starting points. Then, each new point generates a geometric vector and a color vector, which pulls the values of the point in a certain direction (for example, further from planet center or more red).
    - Brownian motion - at each new level, the new mesh is linearly interpolated from the previous, and each point moves in a random direction by an arbitrary unit amount. Also, each color changes by some color vector of an arbitrary amount.
    - Gaussian - The new mesh of a level is generated by

    - Random feature creation - Using the characteristics of the sphere, detail is rendered to match the overall geometry and color scheme. For example, if zooming in on a continent of a planet with only green land and blue sea, perhaps a lake will generate, as the sea is registered as a type of feature elsewhere in the mesh.
- Levels of detail - The user is able to adjust the level of detail, which is the amounts of thresholds.

Attempting to dynamically subdivide a Three.js mesh

Initially, the plan to make zoom functionality was to preprocess each level of detail the first time the user zooms in enough to trigger that level. Then, the new vertices, faces, and texture coordinates would be saved in an object, and each level's corresponding object would be saved in a list dedicated to preprocessed resources. This would be computationally efficient but memory inefficient. However, we could not do it this way because of the way Three.js handles geometry. We attempted to use a Geometry object (PlaneGeometry to start), but the problem we ran into was that the mesh doesn't support a change in the number of vertices to render after initialization. We tried to use PlaneBufferGeometry, but the issue with our algorithm now is that we cannot iterate through the vertices and faces because the object has buffers of some vertices and faces, not lists of all of them. Unfortunately, this meant that we had to alter our approach. We decided to switch preprocessing to instead save a SphereBufferGeometry at each level, with appropriate level of detail, and render the appropriate sphere at each step.

Space Background

In order to implement the space background, we first set the background of the scene to an off-black color. To simulate stars, we placed 400 white cubes of varying size (with flat shading) randomly around a sphere of 100-units in radius, centered at the origin. The random-point-in-sphere algorithm used in the implementation was copied from a post on StackOverflow. (link: https://stackoverflow.com/a/15048260)

Lighting

Attempting to add shading to a custom mesh using Three.js has proved difficult. Despite changing the material from BasicMaterial to anything else, the shape that was generated invariably has flat shading, so it still appears as if it were a BasicMaterial. We know it is not a problem with the lights, since when flat shading is turned off for the "stars," the white cubes have darker bottoms. And it is not a problem of not calculating face/vertex normals, or setting the mesh.needsUpdate field to true. As of the submission time of this project, we have not figured out how to fix this issue.

Results

Primarily because of early setbacks caused by unfamiliarity with Three.js and the most efficient way to perform some tasks, we do not have good takeaway results as of now. Please refer to the Readme for a current list of what features are most useful and successful in our implementation.