Youssef Bouhadiba & Zeus Borrego

CS40 Final Project Write Up

Ray-Marched Infinite Terrain Generator using Value Noise

For our final project, we decided to implement an infinite terrain generator using fractional brownian motion and noise functions to generate the terrain and raymarching to be able to see the terrain. When loaded, the code generates a camera that automatically pans and moves throughout the scene, casting rays to gather light information and color each pixel appropriately. The program also creates a moving light source that follows and orbits the camera, the impact of which is calculated by phong lighting with shadow rendering support. Finally, multiple Signed Distance Functions (SDFs) are created which render some primitives that we use to create our terrain. Our TerrainSDF, a modified planeSDF, is passed a point (p) that is manipulated with some value noise functions as well as some fractional brownian motion functions to vary its surface. We detect if the y-value of the point p is within a range and then set the color of it to either sand, grass, or snow, to ultimately create a fairly natural-looking landscape, with water created via a secondary planeSDF without any noise applied.

Our primary inspiration for this project came from Inigo Quilez and his various terrain generator demos on Shadertoy. Being able to see how he was able to accomplish creating such a realistic scene, such as the “Rainforest” demo here, gave us the idea of attempting to implement our own scene using the raymarcher. In addition to Inigo Quilez’s works, we wanted to add some aspects of computer

graphics that we were both familiar with: video games - more specifically, the game Minecraft, which uses noise-based procedural terrain generation as a core gameplay mechanic. Being avid video game players, and since we were rendering our terrain from code, we wanted to apply a method that we were familiar with. Our thoughts immediately went to procedurally generated terrain, which uses algorithms to create and render terrain, as opposed to manually hand-crafting terrain. Inigo Quilez expands upon this greatly in his work, creating extremely impressive and believable terrain, but our goal was more tame. We wanted to combine what we had learned from our raymarcher midterm project with the value noise functions from class to create a custom scene of procedurally generated terrain.

Our raymarcher, as developed for our midterm project, relies on our sceneSDF function to calculate the distance to the p in our scene and the material that it has been set too. The vec3 *eye* of our camera and vec3 *worldDir* variables are passed into our raymarcher, where it will then loop over a certain amount of times, calling our sceneSDF function each time. SceneSDF will return a vec2 type containing a float indicating what material type it is - determining the color values for ambient, diffuse, and specular light - and a float indicating the distance to that surface material from the camera *eye*. Our raymarch function then determines if this distance is too great or not and will return a vec2 type. This is saved as *dist* which we then use to determine if the

point p is greater than a MAX_DIST that we set, in which case we set the fragColor to be our background color - in this case, sky blue.

The y component of our variable *dist*, representing the Material ID, is then passed into our getMaterial function, the output of which we save into a variable called distMat. The getMaterial function determines what the materials of the point should be by use of the material ID that was given in our sceneSDF call. This is then passed into our lighting function phongLighting, along with the coordinates of the camera eye and our point. The phongLighting function essentially determines the lighting behavior and whether or not the point passed into it is “in shadow” through a helper function called phongContrib. The function phongContrib in turns calls a helper function named inShadow, which then returns true if the absolute value of the distance between our light source and the point p is greater than a certain amount, and false in the case that it isn’t - determining if the light is occluded at point p by another object. The function inShadow returns this value to phongContrib which will set diffusion and specular values of the point to a certain value depending on the bool returned by inShadow. Once phongContrib’s call ends, it returns a vec3 to phongLighting that phongLighting returns back to our main function, which we set as our color variable.

Other concepts that we employed in our code were creating a camera that we could position in the scene, phong lighting with shadows and a material ID system. All

of these concepts were used in previous lab assignments and we found it to be more interesting to combine all of them together to create a more advanced terrain generator. Our camera system takes in 3 different vectors to determine how to orientate itself in the scene: a vec3 *eye* which is where we position the actual “eye” that we will see out of, a vec3 *at* which is where the camera will be looking in the scene, and a vec3 *up* which is what the camera should consider to be the vertical direction in the eye frame. Using this camera system allows us to see and move freely throughout the scene, which we ultimately ended up doing. Instead of having our vec3 *eye* variable taking in static floats, which would position it in a fixed position, we used a time uniform named *iTime* as the x component of the *eye* vector, keeping the y and z components constant.

The lighting we have for our scene is based on the Blinn-Phong model of illumination, which allowed us to have support for shadows and ambient, diffuse, and specular lighting that interacts convincingly with light sources. Our implementation of Phong lighting includes a materials system in order to define the reflective properties of each object, specifically in the form of vec3 variables representing the RGB values of ambient, diffuse, and specular lighting. Each rendered object is given a material ID that corresponds to a predefined material, which is passed into the phongLighting function to calculate the color of each point. The phongLighting function returns a vec3 type which we set our *color* variable equal to and pass it on to our vec4 *fragColor* variable. Our decision to use this method of lighting came down to our familiarity with it. The

midterm project done earlier in the semester made us comfortable with how it should be implemented, but more importantly gave us the experience necessary to to be able to use it again in a different way.

Our entire project relies heavily upon value noise functions, which were gathered from Inigo Quilez's own projects. Our first implementation of noise functions, using exclusively two-dimensional noise functions, left us with a generated terrain that was too grid-like, often leaving very visible lines across the x and z axis. While we understood how our terrain would be generated, we struggled primarily with getting reasonably realistic terrain. After some experimentation with layering multiple two-dimensional noise functions atop each other, and then referring to several terrain generator demos on Shadertoy - and especially those of Inigo Quilez - we realized that instead using a single three-dimensional noise function would lead to greater improvements in terrain realism over any two-dimensional functions, without the grids-and-lines issue that we had encountered previously. Intuitively, this makes sense - adding another dimension of obfuscation to a noise function would remove such visible two-dimensional artifacts like lines going directly in the x- or z-axis.

While overall our work does reach a mostly satisfactory result, creating terrain that appears somewhat realistic and varied, it ultimately fell somewhat short of our expectations in some aspects, largely due to time constraints. Some of the largest

issues that we ran into when starting this project was familiarizing ourselves with the way that the noise functions work and getting the planeSDF to map the noise values in order to change the height of the plane. Originally, the plane that we would generate would be too noisy, causing severe efficiency problems. To fix this, we used some easing functions which would add some smoothing to the connections between the height values generated. However, as our terrain became smoothed out we realized that grid like patterns would keep appearing in the terrain leading to us eventually switching out our 2D noise functions for ones developed by Inigo Quilez.

One goal we wished to achieve early on, and in fact intended to do as our first task, was adding support for soft shadows, rather than the simple binary hard-shadows as we had already implemented in the midterm raymarcher. However, we ran into great difficulty attempting to implement soft shadows, and ultimately decided to prioritize implementing our noise functions and creating rudimentary terrain first before re-attempting soft shadows. Time constraints unfortunately led to us being unable to try implementing soft shadows again, so the code currently only supports hard shadows.

Another feature we planned to implement, but were unable to, was texture support to allow more realistic materials and terrain. For this feature, we had hoped to be able to import several terrain textures to improve our materials like grass, sand, and snow, which otherwise would receive their color and realism solely from Phong lighting values, and we planned to work on this feature relatively late, after our actual terrain

generation was satisfactory. However, we underestimated the time it would take to develop the terrain generation and other more-pressing features, exacerbated by external delays and time-losses, and thus we were ultimately unable to even attempt to implement texture support. If we had more time, this feature would be of high priority, as the current iteration of our terrain generator would likely be improved enormously realism-wise if the materials were textured appropriately, rather than solely being colored as flat Phong-lit objects.

A related feature we failed to implement was the creation of a color gradient between different material layers, in order to make a more natural-looking blend of colors between, for example, the snow material layer and the grass layer below it, rather than simply having a clear borderline between each layer. While we received help from the professor on this issue, we were ultimately unable to get it working - our attempts at easing and blending the materials simply turned all the terrain into a brown blob rather than a natural transition from green to white. We understood the principle of how this feature could be implemented, and re-developed our getMaterial function to support intermediate material IDs to do so, but we were unable to determine how to successfully blend the materials without them being completely mixed together, consuming the defining colors entirely. Additionally, this blending somehow led to significant instability in the terrain, with the terrain shifting and changing shape like a viscous liquid, and bizarre noisy ‘clouds’ of glitchy turquoise generating over the terrain. As time constraints grew, we decided to comment out the code for our

attempts at this feature, and reverted to the hard height-based borders between materials.

A fourth primary goal we had initially intended to reach was to add a free-moving first-person camera, preferably controlled with the mouse and keyboard as in a first person video game, to allow free exploration of the generated terrain. Across the duration of this semester, we have continuously struggled somewhat with camera logic, and eventually we decided to re-define our intended camera functionality to be simpler and more time-efficient to add, as the camera was not the central focus of this project. Thus, we instead implemented a camera that would automatically move forward in order to continually reveal new terrain, with control of the camera's direction being given to the user via GUI sliders.

While less directly quantifiable, another aspect of this project that we would have liked to expand was the complexity of the terrain generator itself. We had initially hoped to generate far more detailed terrain, with significantly higher peaks and generally larger and more varied terrain. Currently, our program creates terrain that appears rather small and relatively flat, as though it were islands and continents on a map or seen from a distant airplane. By contrast, the various demos we had seen on Shadertoy and had been inspired by creates much larger and more detailed terrain, with far more visible noise and detail to landmasses, creating massive craggy mountains with jagged peaks and cliffs, and deep grassy valleys that the camera flies through. We had intended to spend more time experimenting with noise functions and

layering more complexity onto our terrainSDF function in order to approach the level of complexity the Shadertoy demos depicted, however time constraints led to us only implementing a fairly basic set of noise functions for the terrain, resulting in our generated terrain being smaller and lower-detail.

In conclusion, although we did not fully achieve the goals and expectations we had when beginning this project, we ultimately succeeded in creating a value noise-based raymarched infinite terrain generator that creates reasonably realistic and varied terrain. The central premise of what we intended to create was achieved, and in the process we learned greatly about noise functions, raymarching rendering techniques, and the Phong lighting model. Furthermore, we believe we have learned enough in the course of this project that, had we more time, we would have definitely been able to successfully expand the project to reach the initial goals and hopes we had outlined in our proposal, approaching the quality of the Shadertoy demos we had been referencing.

References:

Inigo Quilez's introductory article/tutorial on raymarching terrain:

https://www.iquilezles.org/www/articles/terrainmarching/terrainmarching.htm

Inigo Quilez's article on soft shadows - our unsuccessful attempt at implementation was based on this:

https://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm

"Rainforest" demo by Inigo Quilez - several noise functions we used were derived from this, as well as referencing for general guidance:

https://www.shadertoy.com/view/4ttSWf

"Mountains" demo by Dave Hoskins - also referenced for guidance:

https://www.shadertoy.com/view/4slGD4

Scratchapixel articles on Value & Perlin noise - referenced for developing and using noise functions:

https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1