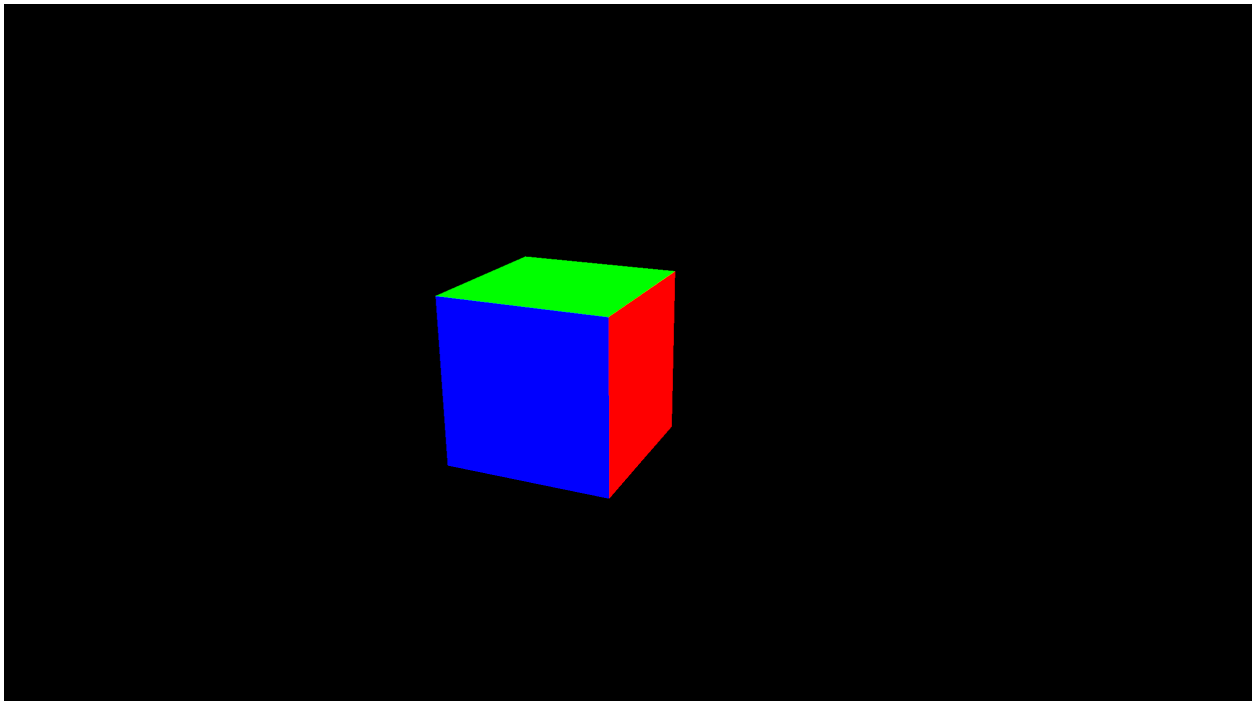
Griffin Dunaif

VoxwellEngine

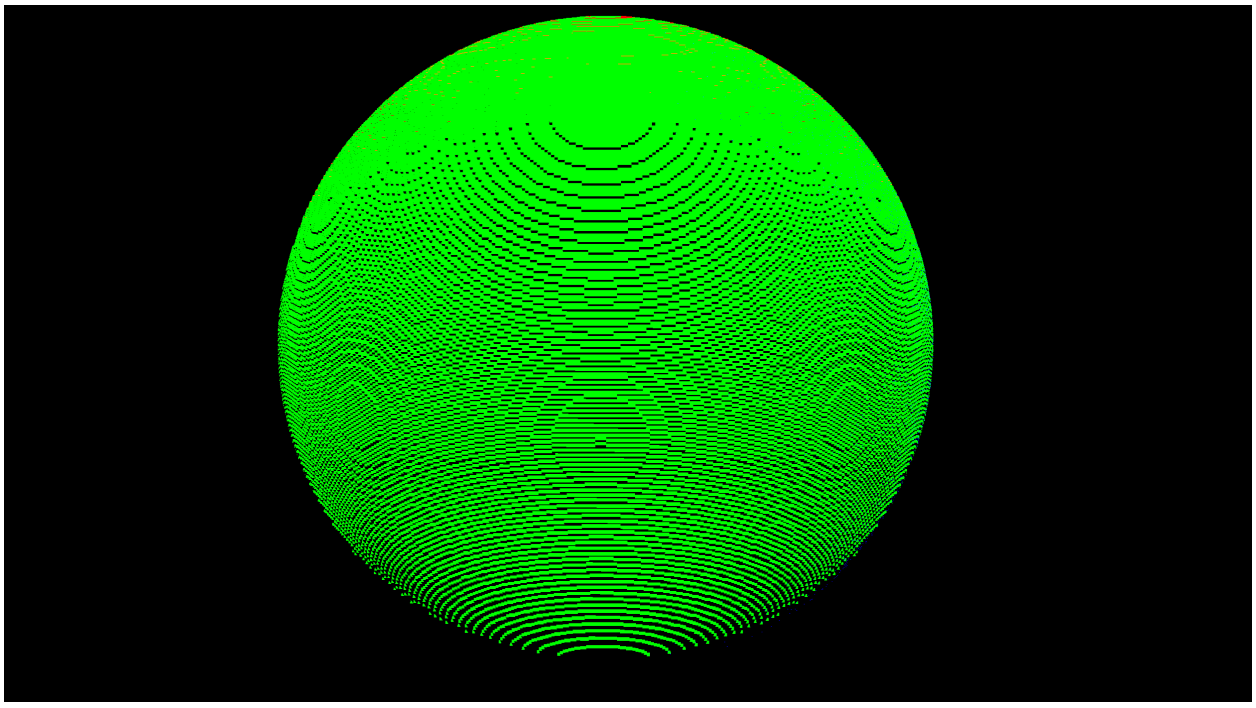
I created a 3D voxel engine that generates procedural worlds by rendering large quantities of 3D cubes to construct interesting shapes and volumes. This is done by mapping arbitrarily complex level sets to cube volumes. That is, each voxel samples from an $R^3 -> R$ function with its 3D coordinate to determine if it should be visible. In other words, if the sampled value is above a threshold value x, where $x \in R$, then the voxel will be displayed.

I began work on Voxwell engine by repurposing a 3D renderer I started working on in high school called MaxwellEngine. It implements a rudimentary rendering scheme in OpenGL (when I originally built MaxwellEngine I followed the learnopengl tutorial and adapted the camera code and other boilerplate OpenGL function calls -- shader, texture, VBO, and VAO creation -- into my own object oriented renderer). I wanted the rendering objects to be as lightweight as possible because VoxwellEngine is supposed to render a large collection of cubes. A game object or actor in Unity or Unreal engine would not be suitable for such a task. These objects come with too much bloat. Thus, I decided to build off of MaxwellEngine and create my own custom voxel renderer. To start, I created a voxel class, which is a data structure meant to capture a 1 by 1 by 1 chunk of the world at a specific coordinate v in world space (world space is a space defined by three orthogonal unit vectors pointing in the x, y, and z directions).



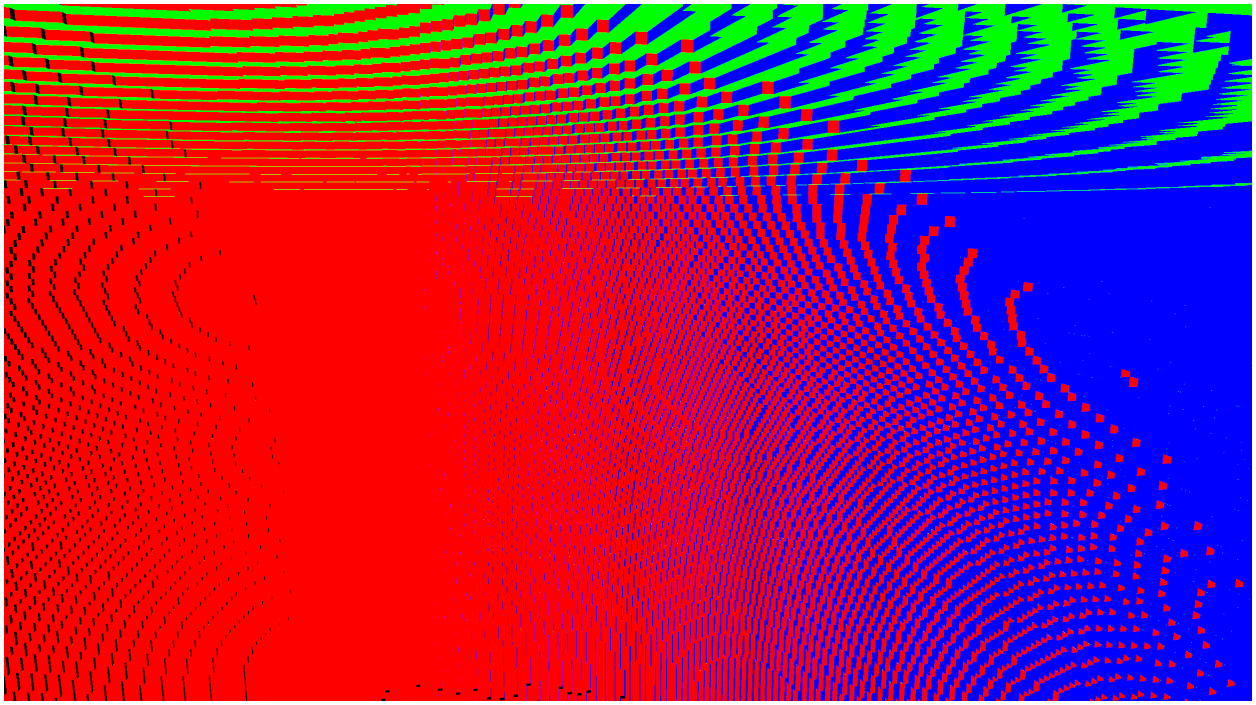
The above image shows a single voxel being rendered at world space coordinates (0, 0, 0). Since many voxels are to be shown on the screen at once, I implemented an optimization where only faces of the voxel that are visible to the camera are rendered. To accomplish this, I split the voxel into a data structure that contains a collection of individual faces and a single byte variable called _options. The first six bits in the byte determine which face is visible (1 visible, 0 invisible) and the seventh bit determines if the voxel is above or below a cutoff threshold defined by a visibility level set (a visibility level set is a function that determines whether a given voxel at an (x, y, z) coordinate should be visible or not). In practice, this optimization is not utilized when rendering a single voxel that is marked visible, since all sides are conceivably visible. However, this optimization proves very useful when talking about the next level of abstraction: chunks.

Chunks are an x by y by z volume of cubes. The challenging part of implementing chunks is doing it performantly. A chunk that is 256 by 256 by 256 will contain 16,777,216 voxels. If every voxel were rendered as an individual mesh this would absolutely obliterate rendering performance due to large amounts of traffic between the CPU and GPU. Thus, I made an algorithm that combined every visible voxel into a singular mesh called the chunk mesh. Below is a rendering of a 256 by 256 by 256 chunk volume where each voxel is checked against the level set $x^2 + y^2 + z^2 = 128^2$ to determine visibility (everything within the sphere is being shown).



Now, it is not enough to merge everything into a singular mesh. There are 16,777,216 voxels in the chunk volume. However, only a fraction are visible to the camera. That is, only the voxels that lie on the sphere's surface are visible. To account for this, I made an

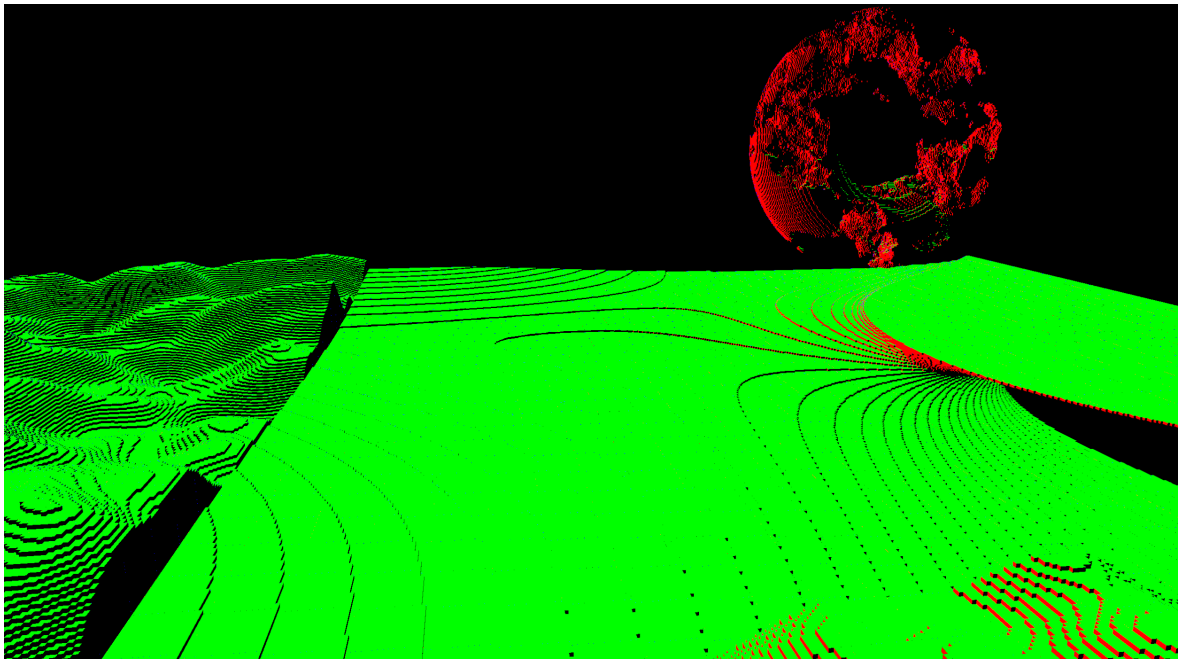
algorithm in the chunk renderer that only added visible voxel faces to one giant chunk mesh (iteratively builds a mesh and sets _options on each voxel accordingly). Thus, only the faces on the surface of the sphere are visible, saving a huge amount of compute. Also, visibility calculations are performed before mesh creation, which is done before rendering. This three step rendering process increases performance, only doing expensive computations when necessary.



Here we can see that the interior of the sphere is hollow, allowing us to render very large volumes efficiently.

Lastly, I implemented another important data structure in my voxel engine called a biome. A biome is a construct that contains an arbitrary 2D matrix of chunks (equivalent to a 2D map when looked at from above). A biome has a location defined by an x and z coordinate. This coordinate represents the bottom-left corner of the biome (the corner with the smallest x value and greatest z value -- greatest z value because the camera points down the -z axis). Also, each biome can map an arbitrary level set to its chunks. I call this level set a space mapping in that it takes in coordinates of voxels in world space, maps them to biome space (a coordinate system whose origin is at the biome location and shares the same unit vector directions as world space), and runs them against a level set (e.g. $x^2 + y^2 + z^2 = 128^2$). It is challenging to map arbitrary functions to voxels due to the hierarchical nature of the voxel engine data structures. A biome contains chunks that contain voxels that contain faces that contain object coordinates. These functions need to be mapped in a way where every voxel coordinate is mapped to the correct location in the function's domain. That is, a level set defined for a biome must map world space coordinates to biome space and then set visibility based

on these mapped coordinates. This allows many biomes to have unique level set functions and be rendered simultaneously in one scene. Here is an example of 4 different biomes being rendered at the same time, each determining voxel visibility using level sets defined in their respective coordinate systems.



To achieve the undulating landscapes and complex patterns. I utilized both perlin noise and interesting level sets. 2D and 3D perlin noise were used to achieve different effects. The torus in the top right used 3D perlin noise along with a torus level set. The hills to the left used 2D perlin noise. And, the overhang in the bottom right used perlin noise with a low frequency. Put this all together and you can create really interesting 3D worlds!