Procedural Voxel World and A* Road Generation

Tyler Loewen, Matthew Kania

1. Introduction

For our final project, we implemented portions of two papers. One of the papers being *Game Engine with 3D Graphics* which details various aspects of the process for procedurally generating worlds constructed out of voxels, as well as some optimization techniques. The second paper, titled *Procedural Generation of Roads* discusses the implementation of a system for procedurally generating roads in varying terrains using the A* pathfinding algorithm. Our implementation was done using Unity and written in C#.

2. Voxel World Generation

2.1. Perlin Noise

The terrain in our world is generated using Perlin Noise (2.12 in Oliveira, 2016). Perlin noise is particularly useful for generating more natural looking terrain features due to it utilizing pseudo-random noise. Pseudo-random noise has gradual changes between intensities, unlike completely random noise. These gradual changes allow the terrain to be generated randomly while still being recognizable as terrain and not random noise.

We create a heightmap by calculating a value between 0 and 1 for each point on the heightmap, each point being the size of a voxel (2.13 in Oliveira, 2016). The calculations for these values are done using a Perlin noise function along with five variables, scale, number of octaves, persistence, lacunarity and a seed.

Multiple octaves are used to add smaller-scale detail to the terrain while preserving its overall shape (e.g. small rocks on a hill that are part of a mountain). The more octaves used, the more varying levels of detail can be achieved. A separate heightmap is generated for each octave using the built-in Perlin noise function where each map is then modified using persistence and lacunarity variables. Persistence is a value between zero and one that decreases the amplitude of each successive octave. It does this by squaring itself every new octave and then multiplying this new persistence value with the noise value at the current point in the heightmap. This has the effect of controlling how rapidly the amplitude decrease between octaves, therefore causing every following octave to have an increasingly smaller effect on the shape of the terrain. The lacunarity variable works in a similar way, except it decreases the wavelength between octaves rather than the amplitude. This causes each octave to have an increasingly larger effect on the variation of the terrain. Therefore, as the frequency of the octaves increases, their amplitudes decrease to reduce the effect the octave has on the shape of the terrain.

The scale variable allows for the scale of the generated Perlin noise to be modified. This is useful for generating terrain with more subtle variations in features by increasing the scale, or for creating drastic changes in terrain using a smaller noise scale. The provided seed allows the Perlin number generator to generate the same number sequence reliably. This means that the exact same generated terrain can be used again.

Together with these variables, it is possible to create a wide variety of terrain styles and features (Figure 1).

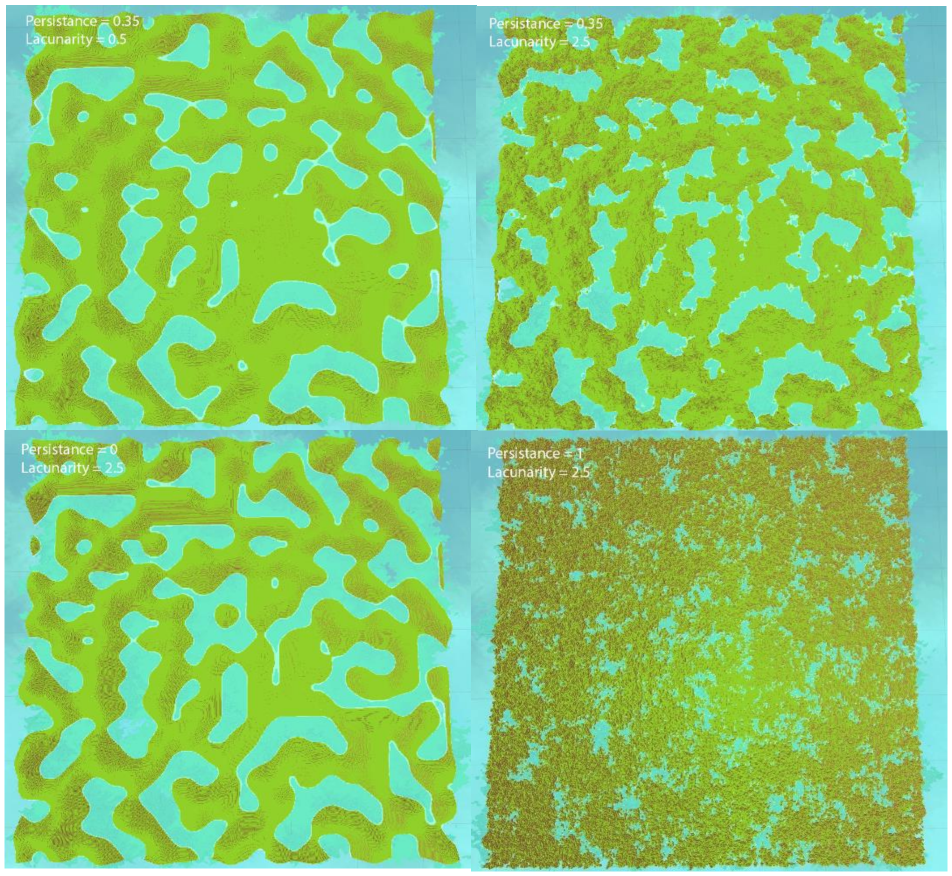

Figure 1: Terrains with varying persistence and lacunarity values

2.2. Voxel Structure and Generation

The generated world consists of a 3-dimensional array of imaginary cubes of a predetermined size known as "chunks". In turn, each of these chunks contains a 3-dimensional array of small cubes referred to as "voxels."

The world is established using a heightmap to fill the voxel array one-by-one, with each point on the heightmap corresponding to one column of voxels in the world. Each column is built by taking the value at a point in the heightmap and using that to determine how many voxels will be instantiated upwards in the y-direction, starting from the base of the world. After the chunks are filled, each chunk then uses a meshing algorithm to convert its voxels into vertices and triangles, which are converted into a mesh that represents that entire chunk. When all chunks have been converted into meshes, the world has been generated

Using chunks to group voxels provides multiple performance improvements. One such improvement is the ability to only recalculate and update the mesh of a single chunk that was modified, instead of needing to recalculate every chunk in the entire world. The second improvement is that view frustum culling can be performed, where chunks are only rendered if they intersect the camera's viewing frustum, preventing the entirety of a large world from being rendered when only a small portion of it is visible. (2.14 in Oliveira, 2016)

2.3. Optimization

Generating a large world consisting of millions of voxels can be an expensive task regarding execution time. Keeping this in mind, optimizations of the chunk meshing process, as well as the overarching world generation process were implemented, significantly reducing both the generation time as well as the resulting number of vertices and triangles used to display the terrain.

2.3.1. Greedy Meshing

To minimize the number of vertices and triangles in each chunk, and thereby optimize both the meshing time of each chunk as well as the rendering time of each frame, we employed a greedy meshing algorithm. When converting the voxels inside each chunk into a mesh, the algorithm runs six times, once for each face direction. Within each run, the algorithm starts at the lowest-left corner of the face and, iterating over each visible voxel, loops over adjacent voxels to find the largest rectangle of identical voxels. Once this largest-rectangle is found, the algorithm returns the corners of this rectangle as vertices for the final chunk mesh. The algorithm continues until it reaches the top-right corner of each face. This results in a mesh with a significantly lower vertex and triangle count than if each individual voxel's vertices and triangles were included in the final chunk mesh (Figure 2). (4.1 in Oliveira, 2016 / Lysenko, 2012)

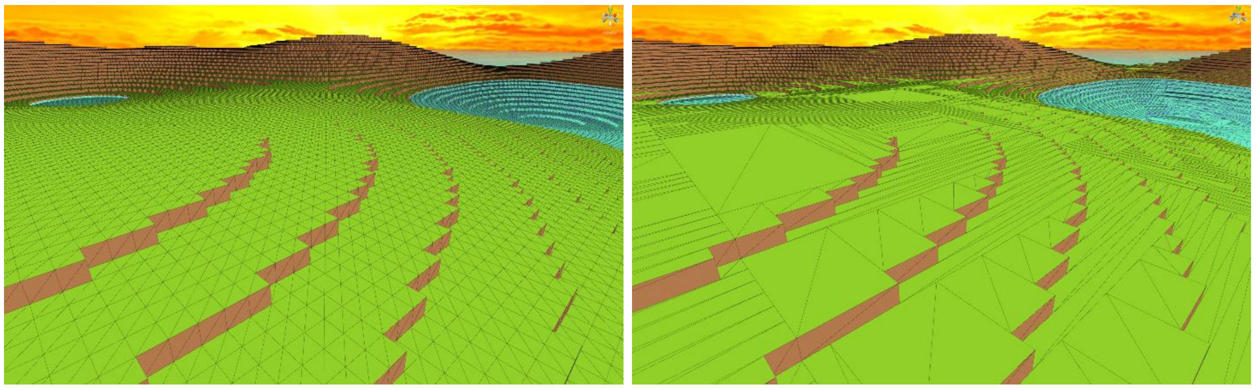

Figure 2: *Naïve meshing (left) and greedy meshing (right).*

2.3.2. Multi-Threading

To significantly reduce the time taken to generate worlds, as well as to prevent the program from locking up until the process is complete, the use of multi-threading was implemented. By creating a pool of threads and passing it the list of chunks, threads could be assigned to individual chunks, performing the necessary code to convert the voxel arrays into a mesh in parallel with the main thread of execution.

2.3.3. Comparison

We ran multiple tests using worlds of different sizes, comparing the time it takes to fully render the terrain with greedy meshing both enabled, and disabled. The same seed was used for every test ensuring identical terrain between tests. The results we found, shown in Table 1, show that greedy meshing is indeed very effective in decreasing the render time of the world. We found that there was an average rendering time saving of 45%. This means that as the size of the terrain increases, so does the amount of time saved by using greedy meshing. The type of terrain did not seem to have a significant impact on the rendering time. (4.1 in Oliveira, 2016)

Chunk Size / World Dimensions	Greedy Meshing	Naïve Meshing	Δt	Vertices (Greedy)	Vertices (Naïve)
32 / 16x3x16	13s	19s	+46%	1,500,000	8,900,000
32 / 24x4x24	42s	60s	+43%	4,300,000	20,600,000

Table 1: *Results of greedy meshing tests.*

3. Road Generation using A* Pathfinding

Following the *Procedural Generation of Roads* paper, A* pathfinding was implemented to calculate road paths between given points. A* is a pathfinding algorithm that searches for the cheapest path between two given points, determined by an "F-cost", which is the sum of a "G-cost" and an "H-cost". The G-cost is the cumulative cost to travel between the starting node and the current node, and the H-cost is the distance between the current node and the target node. To accomplish this, the world must first be divided into evenly-spaced nodes. This step was made trivial by our decision to construct the world out of voxels, which inherently act as nodes. Next, the algorithm creates two empty lists; an "open set" that consists of neighbouring nodes to potentially be explored, and a "closed set" that consists of nodes that have already been explored and evaluated (Figure 3). The algorithm then adds the starting node to the open set and begins its main loop. Within the loop, the algorithm finds the lowest-cost node currently in the open set and moves it to the closed set. It then finds all valid neighbour nodes that are not already part of the closed set, calculating their F-cost and adding them to the open set. The current node is also marked as the parent of the neighbour nodes, which permits backtracking of the final path. The loop continues until either the target node is reached, or no nodes remain to be selected in the open set. If the algorithm successfully reaches the target node, the final path can then be constructed by following the parent nodes from the target node back to the starting node and reversing the list. (Lague, 2014 / 3. in E. Galin, 2010)

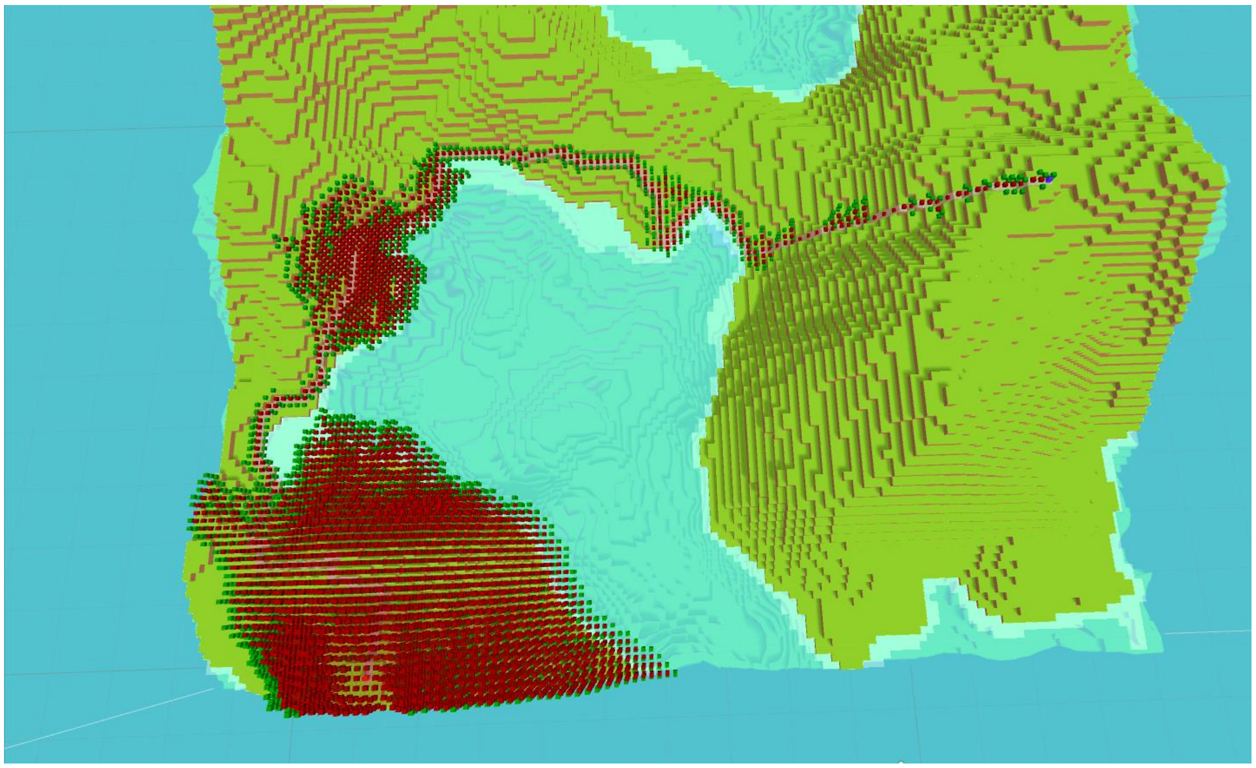

Figure 3: *Complete traversal of a pathfinding instance (Red: Closed Set, Green: Open Set).*

3.1. Customization of the A* Algorithm

Achieving common road features such as slopes, tunnels, and bridges required expanding the basic A* algorithm. As the construction of such features in real life incurs additional costs over simply paving roads across flat fields, real roads tend to travel around, over, or through various features of the raw landscape, depending on what is most cost-effective. To mimic the results of these considerations, we modified the G-cost calculation to consider various notable details about the voxels in neighbouring nodes, such as changes in altitude, sudden cliffs, and walls of dirt. The resulting G-costs help the pathfinding algorithm decide when and where it's cost-effective to build bridges and tunnels, or simply find a flat way around.

3.1.1. Water

To increase the variety of pathfinding scenarios we implemented water which allows for more variation in possible road paths. The water is represented by a single world-spanning quad with a custom shader that roughly simulates the appearance of water by applying distortion and simulating a shoreline based on depth values (Figure 5). This shader was constructed in Shader Forge, a node-based shader editor.

3.1.2. Tunnels

As the road is being built tunnels are created in situations where the cost of going around or over a terrain feature, such as a hill or mountain, has a higher cost than building directly through the feature (Figure 4). Tunnelling features both a variable tunnel radius measured in voxels, as well as a variable G-Cost to specify how expensive the tunnelling operation is. (4.2. and 5.3. In E. Galin, 2010)

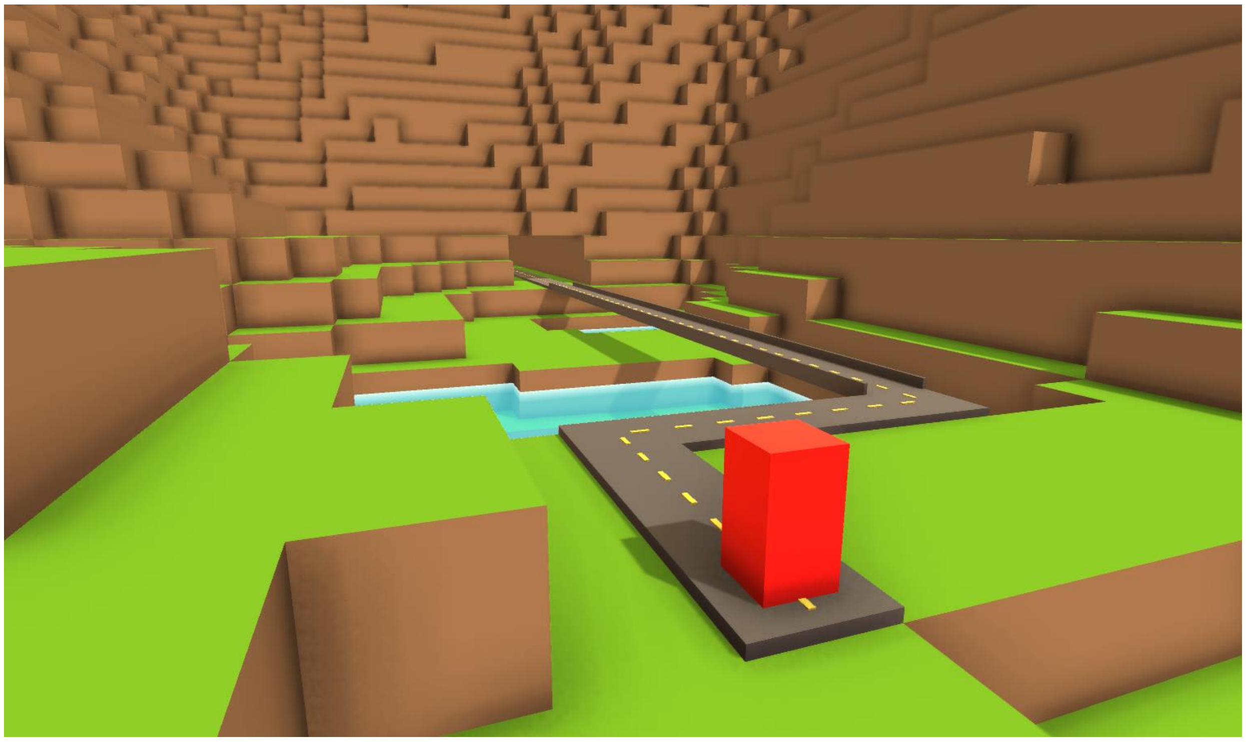

Figure 4: *Road from the starting location creating a tunnel towards the destination*

3.1.3. Bridges

Bridges are constructed over land or water when that is a more cost-effective route compared to going around water or a valley (Figure 5). When the algorithm locates an empty neighbour node with one or more empty nodes (voxels) underneath, there is the option for a bridge to be constructed, rather than attempting to find a way around the cliff. Like tunnels, this feature possesses a variable G-cost, with the cost being multiplied based on the height of the cliff. In situations where the cost of building bridges is high, this may result in short bridges remaining as close as possible to ground or sea level. To the contrary, if bridges are set to have a low cost it can result in long bridges passing over air. (4.2. and 5.3. in E. Galin, 2010)

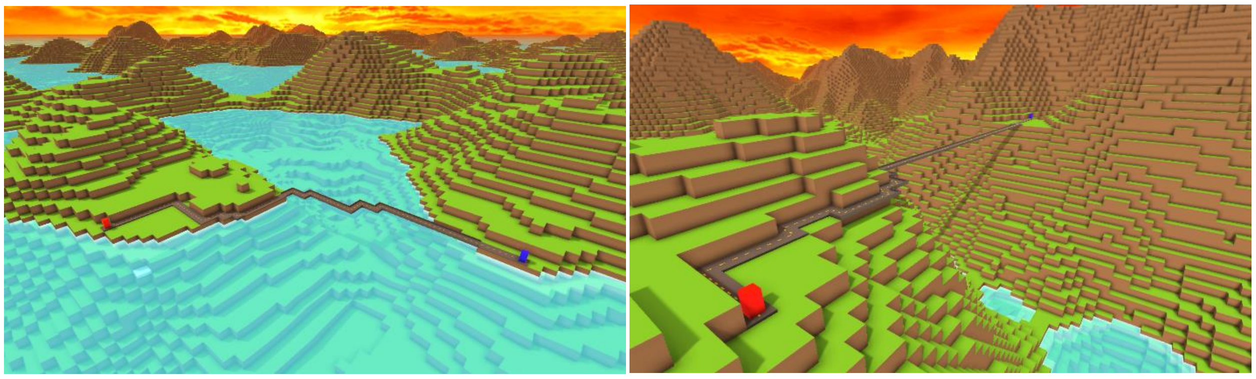

Figure 5: *Bridges across water (left) and land (right).*

3.2. Visualizing the Algorithm

While the path is being calculated, the A* pathfinding algorithms exploration of the world's nodes is visualized in real-time in the Unity editor "Scene" tab. Evaluated nodes in the closed set are represented as red cubes, while potential neighbours in the open set are represented as green cubes (Figure 3). This visualization of the algorithm was invaluable during debugging and bug-fixing of the pathfinding code and serves to help the user better understand the algorithm's decision-making. In addition to being able to view the open and closed set, while the final road is being drawn, a short time interval is introduced between each tile being laid. This short interval allows the construction of the path to be easier seen.

4. Conclusion

By following the implementations discussed in both papers, we could successfully implement our own procedurally generated voxel world along with a pathfinding system that permits for roads to be automatically constructed between two given points. Further, we ensured our implementation possessed minimal execution time and maximum performance by implementing optimizations such as multi-threaded chunk meshing to reduce world generation time, as well as applying a greedy algorithm to the meshing process itself to minimize the number of vertices and triangles belonging to the meshes of each chunk.

Our implementation could certainly be expanded upon, such as implementing different voxel types including mud, rock, snow, and so on, as well as obstacles such as trees or boulders. Another feature that could be implemented is a wider road width. Using wider roads in a voxel world would require determining if any voxels intersect with the road's path. These intersecting voxels would then have to be removed to make room for the road. Empty voxels underneath the road would also have to be checked for, and if found, filled in to support the road. By completing these steps, wider roads could be generated on many different terrain scenarios, like building a road parallel to a "staircase" slope. (6. In E. Galin, 2010)

Regardless of these possible additions, we believe that our current implementation exists as a solid foundation for any system that would require a procedural voxel world with pathfinding capabilities, whether it be terrain modelling software or a video game.

References

Oliveira, Caue Viegas, et al. "Game Engine with 3D Graphics." 2016.
https://pdfs.semanticscholar.org/f269/4330ed2ef9bf2f3db2a3099b6ac8ef222d66.pdf

E. Galin, et al. "Procedural Generation of Roads." 2010.
https://www.researchgate.net/profile/Eric_Guerin2/publication/229707505_Procedural_Generation_of_Roads/links/59b2405c458515a5b48ab3f1/Procedural-Generation-of-Roads.pdf

Mikola Lysenko "Meshing in a Minecraft Game." 2012.
https://0fps.net/2012/06/30/meshing-in-a-minecraft-game

Sebastian Lague "A* Pathfinding." 2014.
https://www.youtube.com/watch?v=-L-WgKMFuhE