A* Agent Traversal on Marching Cubes

Corbyn LaMar, Salem Richie, Skyler Boelts

1 Introduction

Believable terrain generation is integral to almost all procedurally generated video games. Many of these games require this terrain to be modifiable and traversable in real time. One solution for terrain creation is the Marching Cubes algorithm. Marching Cubes is a surface construction algorithm first published in 1987, though it was under patent protection until 2005. When combined with a way to initially generate terrain data, it produces believable voxel approximations of surfaces. This terrain then needs to be traversable, and a simple way to do so is with a grid-based search space and a search algorithm, in this case A*.

This paper details the creation of a proof of concept we made in Unity Engine and C#, along with any issues faced and lessons learned from the implementation of Marching Cubes and of A* pathfinding.

2 Setting Up the Marching Cube Algorithm

The Marching Cubes algorithm builds a mesh by interpreting data associated with points in a scalar field. Our implementation of this algorithm is broken down into the following steps.

2.1 The Marching Table

The Marching Table is a crucial component of the Marching Cubes algorithm. It serves as a static reference table used to determine which triangles should be constructed within each voxel on a regular 3-dimensional grid. The configuration of a particular voxel, corresponding to an entry in the Marching Table, is determined by the values of its vertices (adjacent voxels share four vertices.)

However, in practical implementations of Marching Cubes, due to factors such as interpolation or floating-point errors, there can be ambiguity in determining a voxel's configuration. In these special cases, the values at voxel vertices are not definitively above or below the threshold (inside or outside of the terrain,) resulting in multiple possible interpretations for surface geometry.

For our implementation, we used a Marching Table found online from a student project at an institution in France [Experimenting with Marching Cubes 02]. In doing so, we adopted their understanding of how voxels are configured based on the vertex data within them. This includes accounting for their six found ambiguous cube cases that would normally cause topology problems, for which they provided alternate configurations as fixes.

2.2 Generating the Heightmap

In order to properly generate an interesting surface mesh using the Marching Cubes algorithm, we needed to initialize each vertex with height values. Since we wanted to generate smooth terrain resembling a landscape of rolling hills, the best approach here was using a type of Perlin noise to generate the base heightmap. We generated 3-dimensional Perlin noise by combining multiple 2-dimensional Perlin noise functions along each axis in either direction before normalizing the result. Afterwards, we set all terrain below the generated height for the given y-value to zero, which represents filled-in ground.

Listing 1 Function to generate 3D Perlin Noise.

```
public float PerlinNoise3D(float x, float y, float z)
{
    float ab = Mathf.PerlinNoise(x, y);
    float bc = Mathf.PerlinNoise(y, z);
    float ac = Mathf.PerlinNoise(x, z);

    float ba = Mathf.PerlinNoise(y, x);
    float cb = Mathf.PerlinNoise(z, y);
    float ca = Mathf.PerlinNoise(z, x);

    float abc = (ab + bc + ac + ba + cb + ca);

    return abc / 6.0f;
}
```

2.3 Marching Cube Algorithm

With each vertex now containing proper data for its height, we could verify our implementation of the Marching Cubes algorithm. We loop through each voxel within the grid and note the height at each of its eight vertices using the height values previously generated. If a vertex is above a specified height threshold, then it is considered filled for that voxel's configuration and will influence part of the mesh. Otherwise, a vertex is kept empty for that voxel's configuration as it falls outside of the mesh.

We then use the configuration derived from the values of all eight vertices to determine the indices of the Marching Table which will inform the generation of triangles within the voxel. Generating the mesh within the Unity Engine was simple since our algorithm collected all the resulting vertices and triangles that we could use to recalculate the normals of the mesh and then assign to a shared mesh component.

To smooth out the generated surface, we calculated the position of each vertex (within the triangles composing the mesh) along the corresponding voxel's edge by performing linear interpolation between the two endpoints of the edge. The interpolation factor is determined by the difference between the height threshold and the scalar field value at one endpoint of the edge, divided by the difference between the scalar field values at both endpoints of the edge. This ensures each vertex is placed corresponding to its relative height within the cube.

3 Terrain Modification

With terrain successfully being generated using the Marching Cubes algorithm, we moved on to the next step: modifying the terrain in real time. To start, we ran the Marching Cubes algorithm to generate a new mesh for each frame, which was passable since our grid space was still on the small end at this point. We set up a simple spherical player controller that flies around the space, adding terrain within the sphere on left mouse click, and removing terrain within the sphere on right mouse click. This was achieved by adding to or subtracting from the heightmap for each voxel vertex within our grid.

After implementing the terraforming feature, we also created a flag for the terrain named "isDirty." The purpose of this flag was to ensure that the terrain would not update unless the terrain was modified, minimizing time spent running the Marching Cubes algorithm. If the terrain is modified by the aforementioned player controller, the isDirty flag will be set so that the next frame update would show the changes.

4 Generating in Chunks

While we could generate the terrain now relatively quickly, the speed at which it updated remains dependent on the size of the voxel grid. The larger the grid, the longer it takes to generate initially and the longer it takes to modify terrain with user input. To solve this problem, we needed to scale down the grid structure for a given marching cube script and have multiple grids generated side by side in chunks.

Each chunk handles its own grid and mesh, though a Chunk Manager was written to manage chunks as one unit and communicate between them. When terraforming one chunk, the chunk manager updates that chunk and any adjacent chunks that may have changed. This is to ensure that the edges along the terraformed chunk match. With this approach, the maximum number of chunks being updated at once is nine, though, more realistically, it is going to be between one and four chunks. This runs much faster than regenerating the entire space as one grid, though the optimizations are less noticeable with smaller map sizes. This cross-chunk communication ended up also being important for updating the valid neighbors for traversable nodes across chunk borders.

5 Agent Traversability Map

After building a world with formable terrain, our next step was to allow an agent to pathfind on it. We opted to use a grid-based search structure as we were already storing data in equivalent structures within the chunks.

Creating a list of known traversable nodes would not only help in ensuring the agent does not walk through walls or beneath terrain, but it also helps prune the search space by requiring that only traversable nodes can be considered valid neighbors to other traversable nodes. By verifying traversable nodes as they are modified, we can speed up the search when agents start pathfinding.

The main challenge was determining which conditions make a particular position traversable. For our definition of a valid, traversable position, we did a simple solid check for a given vertex and the two positions above it. The vertex at a given position would need to be solid for the agent to stand on, whereas the two positions above it would need to be empty for the agent to be able to safely stand at that point without hitting their head. A vertex being solid is dependent on its height values in the voxel grid used for the Marching Cubes algorithm.

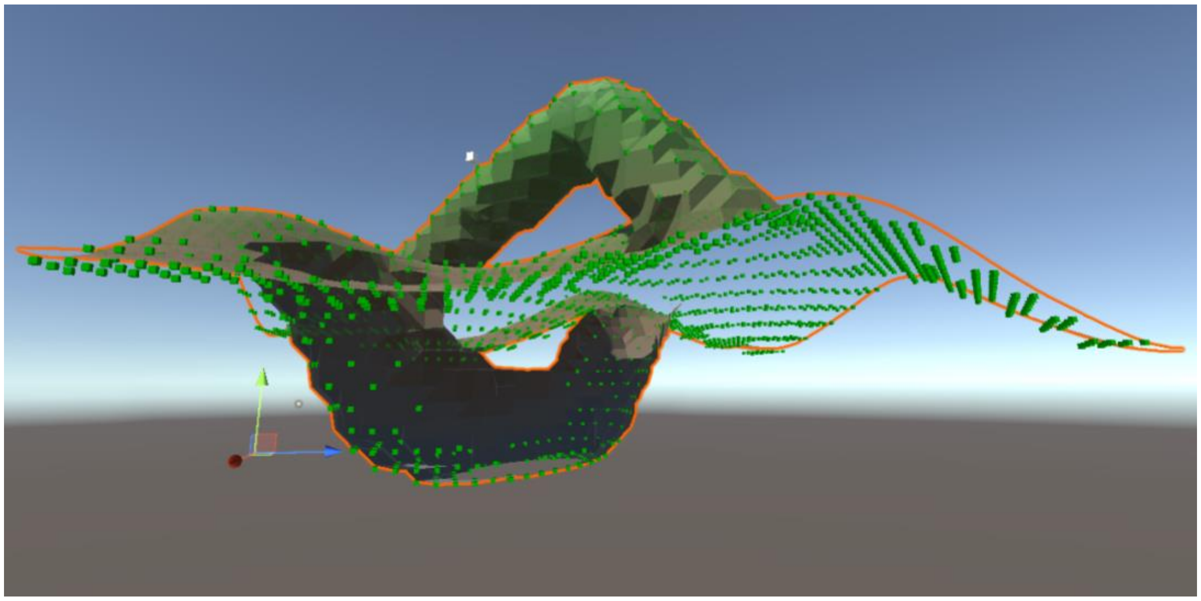
Figure 1 Traversable nodes visualized over the terrain using Unity Engine Gizmos.

6 A* Pathfinding

To get the agent to navigate through the environment, the A* search algorithm is run starting at the position of the agent. A* is a modified version of the Dijkstra search algorithm that estimates the expected cost of paths from discovered nodes to the goal using a heuristic function. The heuristic chosen for the approximate cost was a 3-dimensional Euclidean distance calculation. While other heuristics could be used, this one was used as it is guaranteed to be admissible and not exceed the true cost of a node's remaining path to the goal.

Since traversable nodes and their valid neighbors are recomputed after each terrain change, each node already houses information about its traversable neighbors. Using this data, the agent can reference the traversability map as the search space for the A* algorithm. The pathfinding algorithm references valid neighbors as it adds new nodes to the open list. The open list is organized by cost into range-limited buckets to speed up the pop operation while still allowing for a quick sorted insert operation.

To make the agent movement appear smooth, the agent's position is interpolated between each node on the path. In addition, the agent's y-position is set to match the top facing mesh by performing a raycast downwards to find the exact surface of the mesh and snap to the vertical position of the collision point.

This worked well, but there were some edge cases that came up while testing the pathfinding. As the terrain is modified in real time, it is necessary to ensure that the agent does not walk through invalid terrain. Our solution is to have the agent pathfind again each time modifications finish to ensure it always finds the optimal path.

Another issue occurred when the agent received a path request while being positioned between two nodes. Due to rounding the agent's position to the nearest voxel position, it would sometimes assume the agent was on a non-traversable node. To solve this issue, the agent will check for the nearest traversable node if it is not standing on one. This check has a small radius, so the agent can still be stranded.

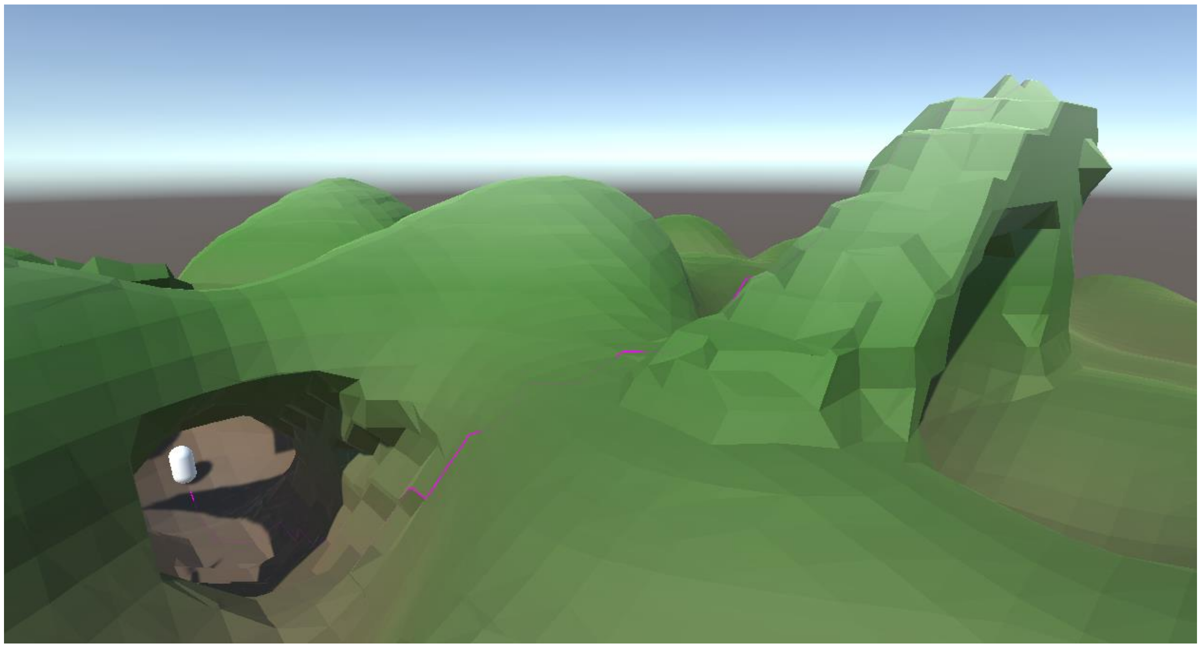

Figure 2 Agent's path visualized in pink using Unity Engine Gizmos.

7 Importance and Application

Combining Marching Cubes with the A* algorithm for dynamic agent terrain traversal offers several advantages in games. Firstly, the ability to have terrain that is capable of being procedurally generated as well as being modified at runtime offers another level of gameplay variety and immersion. Since the Marching Cubes algorithm efficiently converts voxel data into a continuous surface, it enables the creation of diverse and realistic terrain features that can adapt to player actions or external events during gameplay. This is further enhanced by having the agents in the world adapt to these changes.

A* complements this dynamic terrain generation by providing an effective pathfinding solution that allows the agents to navigate through the changing environment intelligently. By continuously updating the traversability graph based on the evolving

Marching Cubes mesh, A* ensures that agents can find optimal paths around newly generated obstacles, altered terrain features, or dynamically shifting regions.

Applying this technique into video games can allow for unique gameplay experiences. For example, players can utilize their tools to dig canyons between their base and an enemy base for protection. Players might also choose to build bridges to allow friendly characters across, or they might create networks of tunnels that non-player agents can still path within. All of this can be achieved by using the Marching Cubes algorithm and A* pathfinding. At a larger scale, appropriate optimizations would be implemented for smoother real time updates to the terrain mesh and A* search space, which are both derived from a 3-dimensional grid.

8 Conclusion

Though this project is not currently optimized or featured enough to be used in a game, it demonstrates sufficient proof of concept for using this style of terrain generation and pathfinding in conjunction. The agent's moment feels believable, and it can pathfind through any sort of terrain made by the user.

Additional features would include optimizations for real time updates to the terrain and additional updates to the traversability map. A variety of terrain brushes and tools would allow for more precise terrain sculpting by players. On top of this, using multiple traversability maps or different heuristic functions would allow different types of AI agents to have unique movement behaviors.

This approach is notable for its scalability and performance. Marching Cubes meshes can represent large and stylized terrains without consuming excessive memory or computational resources. By using portions of the existing grid as a search space for A* pathfinding, this allows for efficient navigation in expansive, flexible game worlds without compromising performance. While not applicable to all games, the combination of Marching Cubes and A* Pathfinding is useful for the right genres within game development.

9 References

References to sourced material for the base Marching Cube algorithm and Marching Tables may be found below.

9.1 Online Documents:

[Experimenting with Marching Cubes 02] Charnoz, Arnaud, et al. Marching Cubes Tutorial, users.polytech.unice.fr/~lingrand/MarchingCubes/applet.html.