View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DigitalCommons@CalPoly

Dynamic Voxel Based Terrain Generation

Thomas Sanford

June 2015

Abstract

This project is an implementation of an editable terrain system. By maintaining an octree of volumetric data and performing the mesh creation on the GPU, the program can allow for free editing of the surroundings which is then reflected in real time. This allows for real time applications to have terrain that can change depending on how the user interacts with it.

Introduction

There are many applications for real-time graphics, but one of the most well-known is in the video game industry. A common theme in video games is the idea that there is a whole virtual world to both explore and interact with. This leads developers to constantly try and create new ways they can present this world. Terrain plays a huge part in that aspect. Whether the setting is a city, a desert, a forest, or a mountain, how the terrain is presented can have a strong impact on the player's immersion. At the same time, terrain is not entirely static in the real world. Given a significant impact, terrain may be moved, deformed, or destroyed. To present the game's world in as much realism as possible, these factors be taken into consideration. This is why it is important to represent the landscape in a way that is not only fast, but is also editable in real time and of a high enough quality to adequately express the setting of the game to the player.

Previous Work

One of the earliest ways of creating terrain has been in the form of a height map. Much like a displacement map, a height map is a top down image that shows the elevation of the map at a given coordinate. While this can be used to generate relatively decent hills or other purely

vertical geometry, it is lacking in the ability to create landscapes that have more than one height at a specific point. For instance, one height value is not sufficient to create caves, tunnels, or any sort of overhang. Another downfall is that vertical elevation is the only variable. While it would be possible to create an editable height map system, only the height at a certain point could change. This is not sufficient if the intention is to have a truly dynamic landscape.

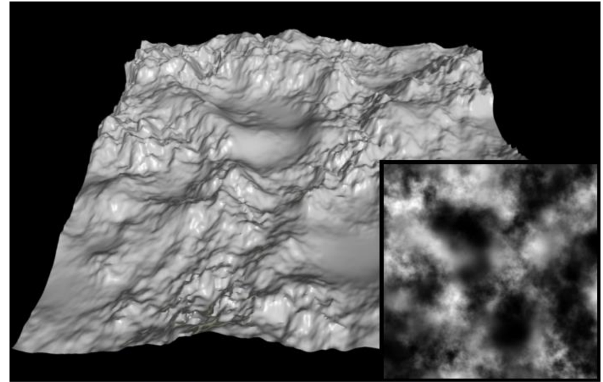

Figure 1: A height map image and the resulting rendered terrain
Images taken and composited from http://en.wikipedia.org/wiki/Heightmap

Another common method of creating terrain is to simply create it as its own 3D model. This method can produce great results, with the upper limit being only the skill of the modeler who created it. It's true that terrain created in this way can look great, but at the same time, it loses the ability to be dynamic in any way. This can be covered for by making pre-scripted destruction animations, but covering all possible cases would require significant time and effort on the part of the asset creators. This method also requires potentially complex collision methods, since the common practice of bounding volumes would not allow for caves or tunnels. In this sense, it is still not an ideal solution.

More recently, people have been looking towards volumetric data as a way to create real time terrain. This data is most commonly represented as a 3D grid of values that can be used to create a surface. Because of this three dimensional grid, it is fairly trivial to compute collisions with the surface. There are also many methods of ultimately representing volumetric data, such as the Marching Cubes [1] algorithm or Dual Contouring [2]. However, even though these methods can create great looking terrain, there is still the issue of edibility. While volumetric data is

intrinsically easier to manipulate in terms of raw data, the actual generation of the final mesh can take a not insignificant amount of time. In order to have truly real time dynamic terrain, the resulting mesh needs to be rebuilt every time the data is edited. This can lead to momentary freezes or skips in the visuals of the game. However, volumetric data on its own has a lot of potential, so it was used as the base for this project.

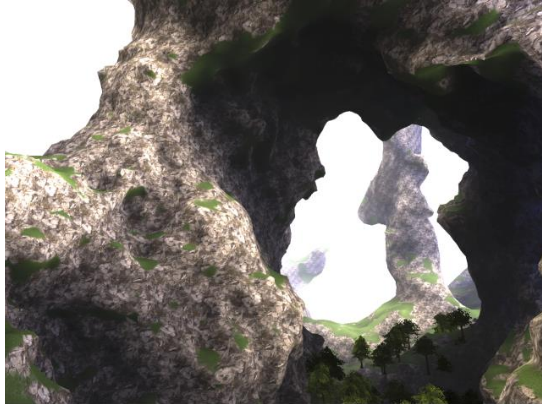

Figure 2: An example of rendered volumetric terrain
Source: http://www.interactive-graphics.de/j_images/VMV2013/results1.png

Algorithm

In order to create dynamic terrain generation, this solution works off of the basic Marching Cubes [1] algorithm. There are a few changes however. Instead of one grid of data, a hybrid tree system is used. Also, all computations for building the mesh fall to a geometry shader on the GPU. By adding these two main steps, it is possible to get all the benefits of volumetric terrain while still allowing for seamless editing.

The general flow of the program follows these steps. First the user creates a new root node. This requires four parameters, a central position, the size of the final grid, the resolution of each leaf, and the maximum space allowed between data points. With this data it recursively creates the octree structure, each node spawning eight more nodes until the desired resolution has been achieved. These final leaves are of a separate class that handles the various functions that affect their data. This octree of chunks is the main spatial data structure for the program. Octrees are ideal for spatial subdivision as it divides cubic areas into finer cubic areas and allows for large irrelevant sections to be ignored with a simple bounding box test. Once the structure is initialized,

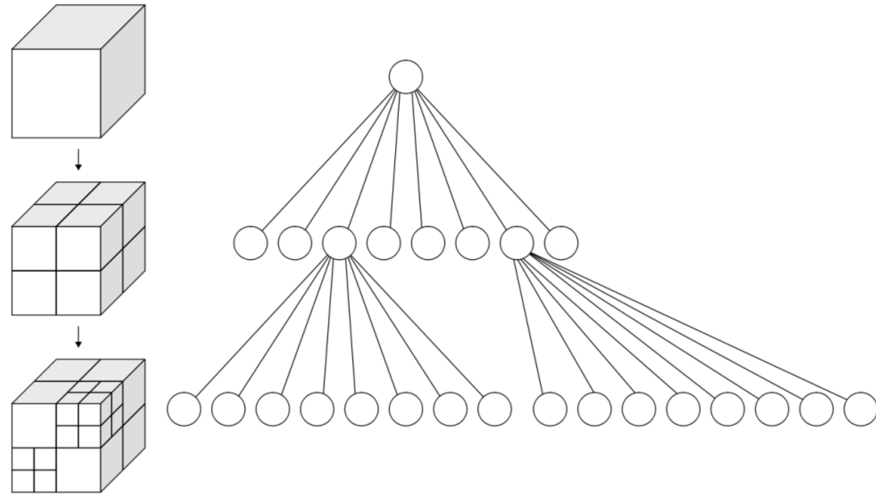

Figure 3: A visual representation of the octree system and how it divides space
Source: http://upload.wikimedia.org/wikipedia/commons/thumb/2/20/Octree2.svg/1280px-Octree2.svg.png

the user is free to add or remove shapes from the data grid. As previously stated, checking the bounding box of the parent node allows for the exclusion of all unnecessary grid tests since the parent completely contains all children. This significantly improves the speed with which the data can be edited and allows for a larger total map size.

At this point, the user has an empty data grid. Data can be added or removed with the appropriate function calls. Depending on the desired shape, the program will use an implicit function to edit the data values in the grid. When removing a shape, only points currently inside a surface are edited, while adding a shape only affects voxels outside of the surface. This allows for the addition and removal of multiple shapes while not affecting any current data. This approach works well with volumetric data since implicit equation results easily lend themselves to isosurface checks.

While adding and removing geometry a user provided parameter dictates whether or not to rebuild the mesh. This allows for mass editing of data if many edits need to happen at once. The act of building the mesh in this scope refers to how the data is packaged for the GPU. This would be the step where the Marching Cubes [1] algorithm would occur, going over each voxel and creating a set of triangles for the resulting mesh. Unfortunately, this leaves the user waiting a noticeable fraction of a second between edits, while also being unable to move the camera.

While this may be acceptable in some uses, such as in a level editor, if the system is to be integrated into an actual game, this is not ideal. To cover for this, the actual mesh is not generated on the CPU. All volumetric data for the leaf is packed into a 3D texture and passed to the GPU, along with 2D textures that contain the required Marching Cubes look up tables and the central position for the chunk. The mesh is then assembled on the GPU by making use of the Marching Cubes [1] algorithm on a geometry shader.

There are many types of shaders, with the most common being the vertex and fragment shader combination. The geometry shader is an addition to this pair, and lies inbetween the vertex shader and fragment shader in the pipeline. While the vertex shader allows for calculations to be performed on existing vertices, the geometry shader allows for the creation of new vertices. This is interesting when considering Marching Cubes [1] since it deals with discrete voxels of uniform size. This allows each voxel cube to be represented as a single central point when making the draw calls. All the resulting geometry can then be created on the GPU,.

Marching Cubes is performed by looking at each voxel in the data grid. Each point has a floating point number that corresponds to its relation to the desired isosurface. The surface in this implementation has an isovalue of zero. This means if the value is negative then it is inside or below the surface while a positive value represents points above or outside of the surface. Using this information the algorithm looks at each set of eight points that create a single voxel cube on the data grid. Using the provided lookup tables, it is possible to find the corresponding mesh segment that should be created given the sign changes

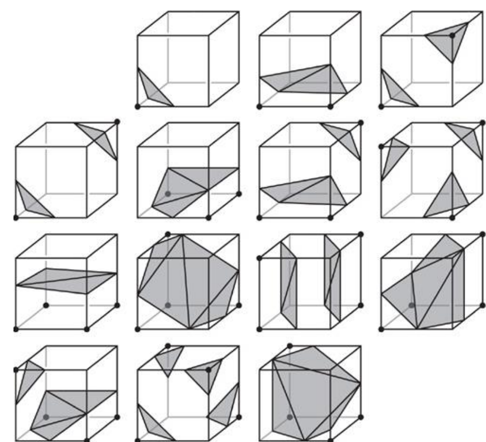

Figure 4: Various potential outcomes of the Marching Cubes algorithm
Source: http://http.developer.nvidia.com/GPUGems3/elementLinks/01fig04.jpg

along the voxel's edges. The geometry shader does this for each voxel in the grid. Because it is performed on the GPU instead of the CPU, this allows the user to not experience any temporary freezes as the only CPU bound computations are simple data edits.

The resulting terrain is also textured. Since a mesh generated from volume data has no natural texture coordinates, they are calculated in the fragment shader. The texture for the material is set to repeating, and three texture lookups are performed for each fragment. The texture coordinates are the xy, yz, and xz coordinates of the fragment in world space. This allows the texture to be projected on to the surface. However, only one color is ultimately needed, not three. To get this, the color is interpolated by using the remaining coordinate value of the normal. For instance, the xy color corresponds to the z coordinate of the normal. Because the normals are unit vectors, the color is then scaled by the square of its corresponding normal value. The final color is given by $(n_z^2 * color(p_x, p_y)) + (n_x^2 * color(p_y, p_z)) + (n_y^2 * color(p_x, p_z))$, where n is the normal vector and p is the world space coordinates of the fragment. This achieves a smooth blend of the texture around the resulting surface.

Results

A small game was created using this algorithm at its core. The player starts at the top level of the grid and has to dig their way down to the bottom. Pressing the space bar excavates a sphere in front of them, while pressing 'c' creates a sphere. There are two material types – sand which makes up the majority of the volume, and stone which serves as an obstacle that cannot be dug through. This was done to give the game a goal as well as show the support for multiple materials that the volumetric data provides.

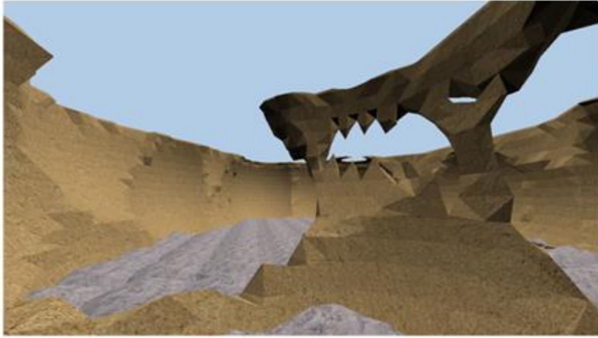

Figure 5

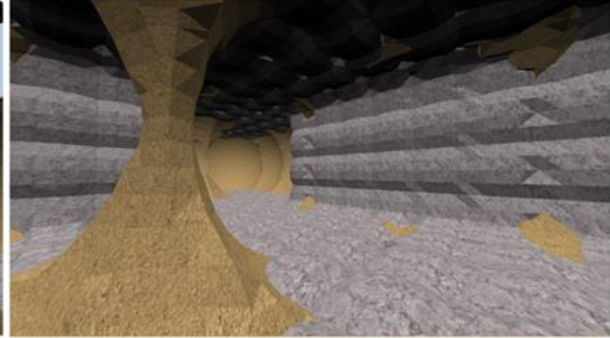

Figure 6

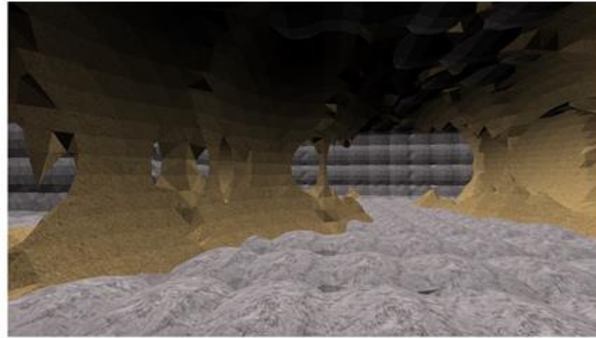

Figure 7

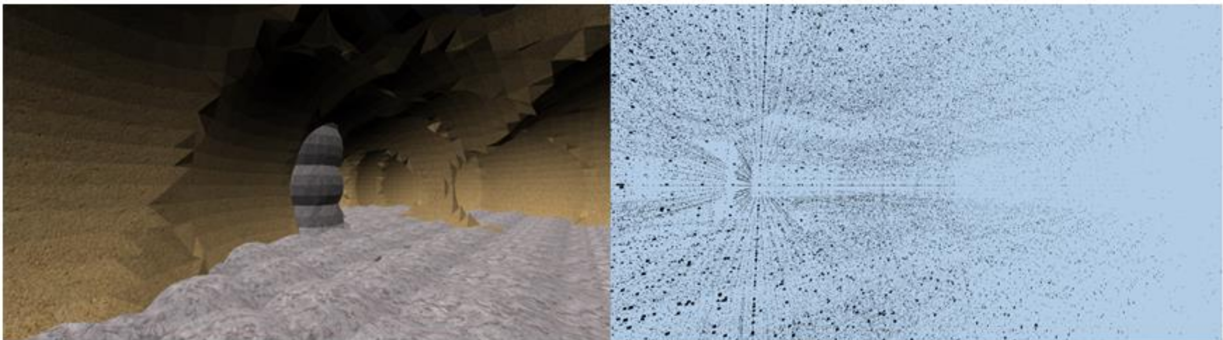

Figure 8: The same view rendered differently. The view on the right shows only the data point cloud represented as quads in space

Future Work

Some future work that could be done on this project would most likely be focused on the initial allocation of memory. While the system itself supports practically any resolution, it is limited by the amount of memory available and the time it takes to allocate it. It is possible to have much better quality of meshes as a result, but it could take anywhere from two to thirty minutes to set up depending on the supplied parameters. The current build starts in only a few

seconds which is satisfactory, but future work may allow for much better resolutions in a similar amount of time.

Another item that could be added is level of detail rendering. The resolution of each chunk does not change during runtime, so all data is sent and processed, even if it ultimately renders to a small part of the screen. By diminishing the resolution of the grid as the camera moves away, it would be possible to see performance increases as less data needs to be processed while maintaining the same graphical quality. The octree system was initially designed with this addition in mind so it would be fairly straight forward to implement.

Conclusion and Limitations

Ultimately, this algorithm is successful at creating an editable terrain system that can run in real time. The octree helps speed up the current implementation, but also leaves room for the algorithm to be expanded. Some limitations exist though. Because the geometry shader builds each section of the mesh in discrete parts, it has no knowledge of vertices outside any one voxel. This leads to the flat shading seen in the images as smooth normal cannot be computed without knowing the neighboring vertices. For a similar reason, shadows are not supported. The common practice of using a shadow map would require the mesh to be built twice per frame, once from the light's perspective and again from the camera's, since the geometry is not static. Outside of these limitations, this is a solid solution for the problem of real time terrain.

Resources

Papers:
[1] Lorensen and Cline 1987. Marching Cubes: A High Resolution 3D Surface Construction Algorithm.

[2] Ju, Losasso, Schaefer, and Warren 2002. Dual Contouring of Hermite Data.

Images:
Figure 1: http://en.wikipedia.org/wiki/Heightmap

Figure 2: http://www.interactive-graphics.de/j_images/VMV2013/results1.png

Figure 3: http://upload.wikimedia.org/wikipedia/commons/thumb/2/20/Octree2.svg/1280px-Octree2.svg.png

Figure 4: http://http.developer.nvidia.com/GPUGems3/elementLinks/01fig04.jpg

General Resources:
Details and Tables for Marching Cubes: http://paulbourke.net/geometry/polygonise/

Specifics on porting Marching Cubes to GPU: http://www.icare3d.org/codes-and-projects/codes/opengl_geometry_shader_marching_cubes.html