Volumetric Rendering of 3D Data

Richard D. Schaefer III

CIS4914, Senior Project
Department of CISE
University of Florida

Adviser: Dave Small, *email*: dts@cise.ufl.edu
Department of CISE
University of Florida, Gainesville, FL

Date of Talk: 7 Dec 2017

Abstract

This report provides a cursory introduction to volumetric rendering and discusses a methodology from both paradigms of volume rendering, Indirect Volume Rendering and Direct Volume Rendering, in depth. The report discusses selected reference implementations of the Marching Cubes Meshing algorithm, an indirect volume rendering technique, and Sparse Voxel Octree Raycasting, a direct volume rendering technique, as well as the developed implementations within the projects environment and those implementations results.

Keywords: Volumetric Volume Rendering Marching Cubes Meshing Sparse Voxel Octree Raycasting Ray SVO Voxel 3D C++ OpenGL

1. Introduction.

Volumetric rendering, displaying a 2D projection of a 3D data set, is a technology of paramount importance for the visualization of 3D data. The volumetric data can be computed, sampled, or modeled and is widely utilized in many different fields. Voxels, volumetric elements, are commonly referred to when discussing volumetric data and are akin to a pixel in 3D space. Within the numerous volume rendering techniques there are two main paradigms and several classifications that further separate the various methodologies. Indirect versus Direct Volume Rendering are two categories of displaying volumetric data with significantly different workflow and results. The Indirect Volume Rendering (IVR) generates a polygonal mesh of the geometry's topology and draws the mesh independently. An obfuscated approach when compared to Direct Volume Rendering (DVR), which draws the voxels corresponding to the volumetric data directly. Beyond these two categories, the first key distinction is whether the rendering algorithm traverses the data, projecting its voxels onto the screen or if the algorithm traverses the screen and calculates the visible geometry for each pixel. Secondly, the voxel representation must be considered; voxels can be stored with a binary or partial representation. A binary voxel is quite simple, it exists or it does not. On the other hand, partial voxels are given a value in between 0 and 1 and the rendering algorithm ignores the voxels below a provided threshold. Many other distinctions exist between rendering methods, derived from the assumptions they make and functionality that they support, but they're neither as fundamental or as pertinent to this paper as the two previously mentioned.

This project entails the research and implementation of established voxel representations and rendering algorithms in the pursuit of dynamically visualizing procedurally generated terrain with C++ and OpenGL. This report documents my trials with both IVR, through the Marching Cubes meshing algorithm, and DVR through both naive rendering and most notably, Raycasting into a Sparse Voxel Octree (SVO). All of the rendering algorithms implemented are tested in an environment procedurally generated with the Libnoise library. The terrain and all necessary data structures are generated at run time when the project is launched and subsequently rendered utilizing the currently selected methodology.

1.1. Problem Domain.

Gathering and visualizing volumetric data has grown to be a critical task in many industries. Medical imaging utilizes volumetric rendering to draw the data acquired from a multitude of scans, ranging from MRI's to ultrasounds. The largest sets of volumetric data are often gathered with geoseismic scans, often requiring the processing of at least 1024^3 voxels. Oil exploration is the most notable application of geoseismic volumetric rendering, the technology can be applied to significantly lower the cost of drilling by selecting an optimal location to do so. Volumetric data is also largely used in scientific simulations and computer graphics. Since each voxel represents a point in 3D space, the technology lends itself nicely to physics simulations. Furthermore, many video games have used the novel properties that voxels provide to create a unique range of games; notable games include *Minecraft* and *Everquest Next,* among a plethora of others. Each domain of use comes with its own requirements and constraints which, through necessity, has fostered significant optimizations and creativity throughout all variations of volumetric rendering.

1.2. Previous Work (Literature Search).

The two most significant documents that were discovered while researching this project are a research paper published by NVIDIA in 2010 documenting their analysis, extensions, and implementations of raycasting with Sparse Voxel Octrees (SVOs) and the original publication of the marching cubes algorithm from 1987. Within both papers lie the implementation details and cognitive basis that were utilized to implement the algorithms and data structures found within this project.

Marching Cubes: A High Resolution 3D Surface Construction Algorithm documents an revolutionary algorithm which would be referenced and built upon numerous times throughout the next decade. The publication explains the process of its namesake algorithm that “creates triangle models of constant density surfaces from 3D medical data” [2]. The algorithm dissects the volumetric data into cubes and uses a series of tables (Figure A1) to approximate a polygonal mesh for each cube, derived from the intersections of the data and the cube. The marching cubes algorithm will be explained in greater detail in the following section documenting the technical approach utilized in this project. Marching Cubes is fast and simple, but with that come drawbacks. Most significantly the algorithm is incapable of rendering sharp features, but can also tend to over-sample simple geometrys and its linear interpolation can often create aliasing artifacts which require additional processing to remove [2].

NVIDIA's research publication analyzes and extends an already well established volume rendering methodology, raycasting, with the goal of storing and rendering large scenes with high fidelity. The paper provides a thorough explanation of their SVO data structure and raycasting implementation, as well as the details and reasoning behind the optimizations applied to both. As with the marching cubes paper, an in depth explanation of the implementation of either data structure or algorithm will be reserved for the following section detailing the technical approach taken within this project. Nevertheless, NVIDIA establishes a SVO structure that emulates the topology of the desired geometry by utilizing child descriptors (i.e. octree nodes) which correlate to each non-leaf voxel [3]. A novel feature in the research paper is the application of a contour for each voxel. The contour constrains a generic cube with a pair of parallel planes that results in "a collection of oriented slabs that define a tight bounding volume for the surface", that can be seen in figure A2 [3]. With the addition of attaching compressed shading attributes to each voxel NVIDIA establishes a robust and efficient data structure for capturing geometry's ad infinitum. The raycasting algorithm implemented by NVIDIA traverses the octree in depth-first order, first calculating the intersection of the ray and the currently selected voxel and ultimately descending the octree until a leaf is found or the ray exits the tree [3]. The implementation can be found in full in the appendix of their paper.

2. Technical Approach (Solution).

This section will discuss the core technologies implemented in this projects endeavor to investigate the various methodologies of rendering volumetric data. The core aspects of the project include the environment generation, the marching cubes implementation and rendering workflow, as well as the generation of SVO's and the subsequent raycasting algorithm.

2.1. Environment Setup.

The main function is within the Game class and is responsible for initializing OpenGL, and creating and managing the Camera object and Chunk Manager in addition to all the state variables that entails; as well as holding the game loop. When initialized, the Chunk Manager is responsible for creating a perlin noise height map via the Libnoise library and using it to generate a N by N array of chunks of size M^3, where N and M are predefined constant integers. An SVO is then recursively generated with the height map and the root is stored. Each chunk stores its respective binary voxels within a 3D boolean array of size M^3. A chunks voxel array is referenced by the marching cubes algorithm when generating a mesh and when naively rendering the chunks voxels. It is important to distinguish that the SVO's data is an entirely separate structure holding only the topology of the terrain. Due to this dynamic, the entire map is generated when the project is launched; N^2 chunks, and a SVO with depth D such that 2^D equals N x M is generated. After this point, the various rendering modes can be cycled through to select how to display the voxels. The voxels can be naively rendered or the marching cubes mesh can be drawn. Further, the SVO can be recursively drawn and individual raycasts can be performed to visualize the raycasting algorithm, or after a full-screen raycast has been computed the voxel set generated can be drawn. The result is a procedurally generated map which can be used to visualize the various volumetric rendering techniques.

2.2. Marching Cubes.

The marching cubes algorithm, an indirect volume rendering technique, is the first key algorithm that will be discussed. In short, the algorithm generates the mesh of each individual chunk by iterating through the voxel array and processing every group of eight voxels, one for each corner of a cube, to determine an approximated mesh based on the groups valid voxels. This collection of polygonal meshes is then stored in a vector and used to draw the topography mesh. In practice, the marching cubes algorithm contains a number of specific generalizations and constructs that create the algorithms efficiency and constraints.

Determining the representation of the current group of voxels is the first step in generating the mesh. A cube, represented by the current voxel and its seven neighbors (Figure 1), is sampled to generate a *cube index*. Each of the eight voxels have two potential states, a voxel is present or it is not. This creates 2^8, or 256, possible configurations of a marching cube, represented by the *cube index*. The *cube index* is then used to index the first significant construct of the algorithm. A key aspect of the marching cubes algorithm is the use of a predefined array to retrieve the edges intersected for each cube configuration. The *edge table* is an array of 256 hex values that correspond to the intersected edges, see figure 2. For example, if vertex 3 is the only valid voxel then our *cube index* will be 2^3, or 8, and the *edge table* indexed at 8 will return the hex value 0x80c, corresponding to 100000001100 in binary, which represents edges 2, 3, and 11, being intersected. For the edge cases when cube is entirely inside or outside of the geometry, the *edge table* returns 0. Once the list of intersected edges

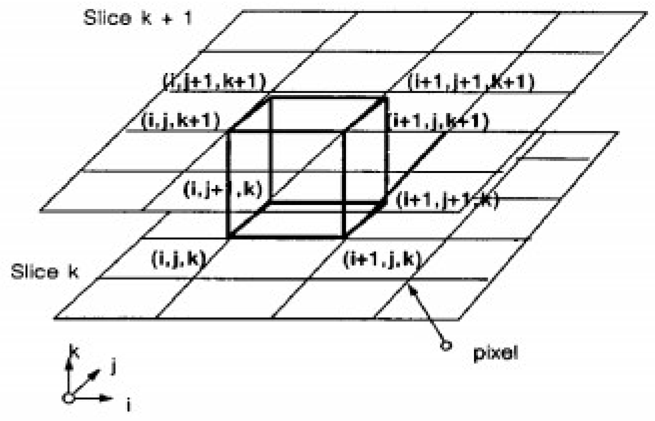


Figure 1: A Marching Cube

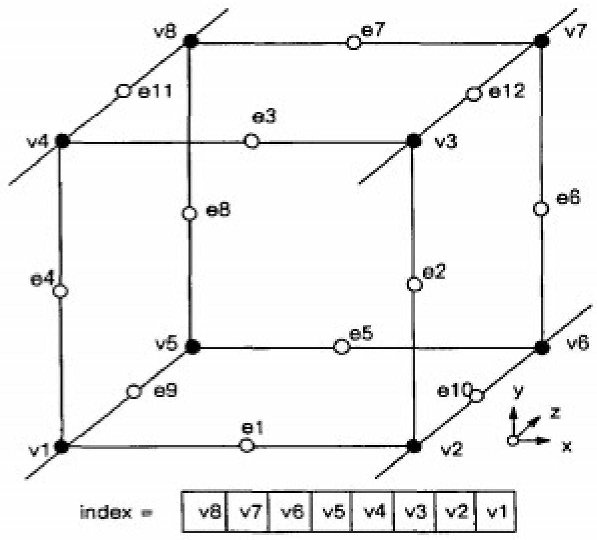


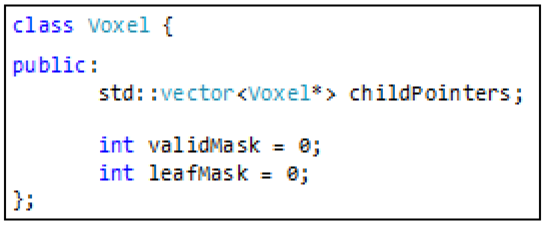
Figure 2: Vertex and Edge numbering.

are acquired, the coordinates of a polygons vertex can be linearly interpolated based on the 'strength' of each voxel in an attempt to approximate smoothness [2]. Since the voxels in this project are binary, the vertex's are created at the midpoint of the intersected edge. After each edge with an intersection has had its vertex approximated the vertices are stored in an array. Triangles are then generated using the algorithms second key construct, a 2-dimensional array referred to as the *tri-table,* and the aforementioned array. The *tri-table* contains the information for which edges vertices should be combined to create the cubes respective mesh for each of the 256 cube possibilities. To continue with the previous example, indexing the *tri-table* at 8 will yield an array of 3, 11, and 2, followed by a series of -1's. Though the entries are often not this simple, a triangle between each of these three edges provides a conveinient visualization. The final step is to calculate the normal vectors and color for each vertex before finally adding each triangle to the chunks rendering array and incrementing the number of triangles within each chunk.

Reconstructing the mesh of a 32x32x32 chunk can take up to 150 ms, a notable delay when seeking responsive terrain. Due to the algorithms O(n) complexity, smaller chunk sizes will result in more responsive voxel manipulations but in turn will require more OpenGL render calls. A healthy medium will have to be devised on a case by case basis. A common anomaly when applying the marching cubes algorithm to a chunk based terrain is the formation of seems between chunks (see figure A3). These result at the boundaries of a chunk where the marching cube would rely on voxels from the adjacent chunk. This implementation has remedied this anomaly by providing each chunk reference to its right and top neighbor upon initialization. When the algorithm attempts to calculate a *cube index* on the boundary of a chunk, its appropriate neighbor is referenced. The marching cubes algorithms is fast, simple, and provides excellent rendering quality within the projects environment, at the cost of initially expensive computation.

2.3. Sparse Voxel Octree Raycasting.

In order to discuss our implementation of SVO raycasting, the octree data structure and its generation must first be explained. A voxel, see figure 3, can have up to eight children and is represented entirely by two integer bitmasks, a *valid mask* and a *leaf mask*, and a list of pointers to each of the voxels valid non-leaf children. Each of the voxels eight children represents an equal subsection of the voxel [3]. The *valid mask* contains one bit per child and represents whether the child slot contains a voxel. Likewise, the *leaf mask* contains one bit per child but represents whether the child is a leaf node.



```
class Voxel {
public:
        std::vector<Voxel*> childPointers;

        int validMask = 0;
        int leafMask = 0;
};
```

Figure 3: The voxel data structure.

To generate a SVO which represents the height maps topology, the current node (initially the root), the height map itself, the current depth, max depth, and the current voxels bounding box represented by two points in 3D space, are passed to a function responsible for recursively generating the current voxels children until the max depth is reached. The function makes calls to construct each of the current voxels child nodes. For a child node to be valid, the height maps value at one of the four corners of the subsection must be within the vertical range of the subsection. If the child node is determined to be valid, a new voxel is created, the current voxels valid mask is updated, and the new voxel is added to the current voxels array of child pointers. If the current depth is not the max depth the function then recursively constructs the children for the nodes which were just created. Otherwise, the function updates the *leaf mask* to represent the *valid mask*. It is of little consequence to the raycasting algorithm but it is important to note that with this representation every leaf is located at max depth and the topology does not cover steep surfaces completely.

The SVO data structure's minimalistic approach shifts much of the size complexity into time complexity in the raycasting algorithm. While size is not an immediate problem within this project, the data size of voxel sets can quickly get out of hand for large scenes. Nevertheless, there are a handful of key variables that make up for the SVO's lack of data, and drive the raycasting algorithm. First and foremost, the current voxel that the ray is traversing is the most important morsel of state. It is represented by the current parent voxel, *parent,* the current child slot index, *idx,* a position vector, *pos,* and an integer, *scale*. These variables largely determine the current state of the raycast and what ultimately gets rendered. The algorithm also depends on a size integer, calculated as the size of a chunk multiplied by the number of chunks, in an attempt to keep the two voxel structures congruent.

Determining the active span of the array is a crucial and repetitive task when comparing the raycast against the SVO. The act of which is only complicated by the necessity of involving a plethora of conditional logic dealing with the signs of the ray direction. The reference raycasting implementation employs a novel alternate coordinate system which avoids the explicit ray direction checks by mirroring the entire octree “to redefine the coordinate system so that each component of d becomes negative” [3]. A feature this implementation lacks. In order to determine the active span of the ray, represented as a *t_min* and *t_max* value, the distance to the intersections of the ray with the near and far planes of each axis of the current voxel are calculated and stored as individual t values. Each t value is then checked to guarantee that it is in front of the origin and within the current voxel. Finally, each t value is compared against *t_min* and *t_max* and if the value is less than or greater than, respectively, the current *t_min* or *t_max* the t value replaces the current respective value.

Once the original *t_min* and *t_max* values have been computed the *idx* is found by comparing the point in space along the ray at *t_min* against the bounding box of the current voxel, and the *pos* is updated. Then until the ray exits the SVO or finds a leaf voxel, the ray traverses the SVO. With each iteration exists three possible avenues for selecting the next voxel. The voxel may descend into a child of the current voxel. The voxel may advance to a sibling voxel, or the voxel may ascend to the "next sibling of the highest ancestor that the ray exits" [3]. The voxel will first attempt to descend to a child if *idx* is a valid voxel according to the *valid mask.* If so, the algorithm checks if the voxel is a leaf and terminates the search or adds the current parent voxel to an array and after updating the *size*, *scale*, *parent*, and *idx,* restarts the loop. If *idx* is not a valid child, the *idx* of the next sibling voxel is calculated. If the *idx* is not valid, meaning the ray has exited the parent voxel, the *scale* is incremented, the *size* is doubled, the *idx* and *pos* of the largest sibling voxel are calculated, and the loop is restarted.

3. Results

Map Generation	*Size: 32*	*Size: 32*	*Size: 64*	*Size: 64*	*Size: 128*	*Size: 128*	*Size: 256*	*Size: 256*
	Octree Generation	Chunk Generation	Octree Generation	Chunk Generation	Octree Generation	Chunk Generation	Octree Generation	Chunk Generation
Trial 1	0.074 s	0.005 s	0.399 s	0.023 s	2.76 s	0.090 s	26.43 s	0.374 s
Trial 2	0.078 s	0.006 s	0.426 s	0.022 s	2.84 s	0.085 s	25.48 s	0.369 s
Trial 3	0.072 s	0.005 s	0.399 s	0.022 s	2.75 s	0.094 s	25.90 s	0.389 s
Average	0.075 s	0.005 s	0.408 s	0.022 s	2.78 s	0.089 s	25.94 s	0.377 s

Mesh Generation	*Chunk Size: 16*	*Chunk Size: 32*	*Chunk Size: 64*
Trial 1	0.035 s	0.138 s	0.619 s
Trial 2	0.034 s	0.141 s	0.617 s
Trial 3	0.036 s	0.135 s	0.591 s
Average	0.035 s	0.138 s	0.609 s
Average/size^3	8.54E-006	4.21E-006	2.32E-006

Each of the rendering methods implemented have very distinct strengths and weaknesses, several of which have been quantified in the present tables. In the *Map Generation* and *Mesh Generation* tables, the chunk generation and marching cubes mesh generation time grows linearly; With each doubling of the map size, resulting in four times the chunk data, the average time required grows by roughly four times. Compared to the octree generation, which increases exponentially with each size increase, due to the fact that an additional layer of depth must be created within the octree to handle the increase in voxel count. In other words, to represent 2^N voxels *N* layers must be present within the octree. This results in an environment where SVOs are difficult to use if the rendering geometry's maximum size increases, but is incredibly useful for efficiently rendering static geometries with high detail. Although the SVO's complexity grows exponentially with size, the raycasting algorithm's complexity grows linearly with each increase of the SVO's maximum depth. This results in relatively significantly more consistent rendering calls when using full-screen raycasting than compared to other volume rendering methods. For example, using SVO raycasting to draw a map of size 32, 32,768 potential voxels, will take ~0.075 seconds to generate the octree and ~27.76 seconds to compute all the necessary raycasts on the CPU, regardless of the number of voxels present. When comparing this to the time complexity of a SVO terrain of size 256, with 16,777,216 potential voxels, it requires ~26 seconds to generate and ~52 seconds to render with raycasting. A 8x fidelity improvement at the cost of 2x the processing power. For a SVO with depth 9 (size 512), holding 134,217,728 potential voxels, an even more egregious example, requires an astounding 813 seconds to generate and only 56.683 seconds to compute the ray results.

Full-screen Raycast Render	*Size: 32*	*Size: 64*	*Size: 128*	*Size: 256*
Trial 1	27.74 s	42.25 s	46.26 s	49.96 s
Trial 2	28.19 s	44.18 s	49.85 s	53.24 s
Trial 3	27.37 s	40.47 s	45.68 s	51.17 s
Average	27.76 s	42.30 s	47.26 s	51.46 s

4. Concluding Remarks

The dynamics of each implemented rendering methodology are insightful to the intrinsic powers of each volumetric rendering paradigm. While naive rendering and marching cubes meshes are flexible and scalable, they fall short when large amounts of data need to be represented. SVO Raycasting relies on a stringent data structure that creates rigidity and introduces complexity to trivial tasks but yields constant rendering times regardless of the amount of data. The strengths and weaknesses of each methodology overlap in such a way that both could be combined to minimize the downsides of the other and create an efficient engine capable of utilizing marching cubes rendering up close and raycasting at a distance.

There is an infinite amount of work that could be done to improve this project. Beyond implementing more rendering methods and further diversifying the the rendering environment to include different scenes, there are a few key limiting aspects of this project. Most significantly, the current method of naively rendering a voxel as a cube reduces the frame rate of large scenes significantly. Employing a greedy meshing algorithm could greatly reduce the number of vertices that need to be rendered without altering the representations appearance. Furthermore, refactoring the construction of SVO's to properly represent steep terrain could provide an appreciable visual boost to the techniques representation.

5. Acknowledgments

The author would like to take this opportunity to thank his adviser, Dave Small, for providing the opportunity and freedom to pursue this project, and guidance, without which this project would've not been possible.

6. References

[1] MeiBner, M., Pfister, H., Westermann, R., Wittenbrink, C. M., "Volume Visualization and Volume Rendering Techniques," cs.unh.edu, 2000, <http://www.cs.unh.edu/~cs880/volvis/Meissner-VolRenderingEGTutorial.pdf> (10 October 2017).

[2] Lorensen, W. E., Cline, H. E., "MARCHING CUBES: A HIGH RESOLUTION 3D SURFACE CONSTRUCTION ALGORITHM ," ACM Digital Library, 4 July 1987, <https://dl.acm.org/citation.cfm?id=37422> (10 October 2017).

[3] Laine, S., Karras, T., "Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation," NVIDIA Technical Report, February 2010, <http://www.nvidia.com/object/nvidia_research_pub_018.html> (10 October 2017).

7. Appendix

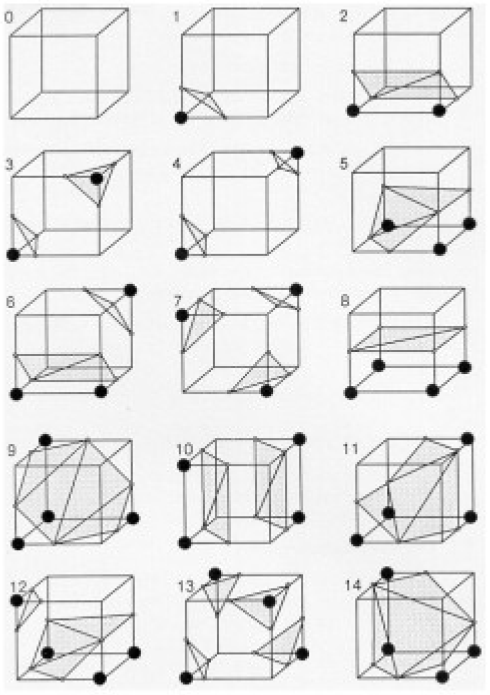


Figure A1: Mesh Approximations [2]

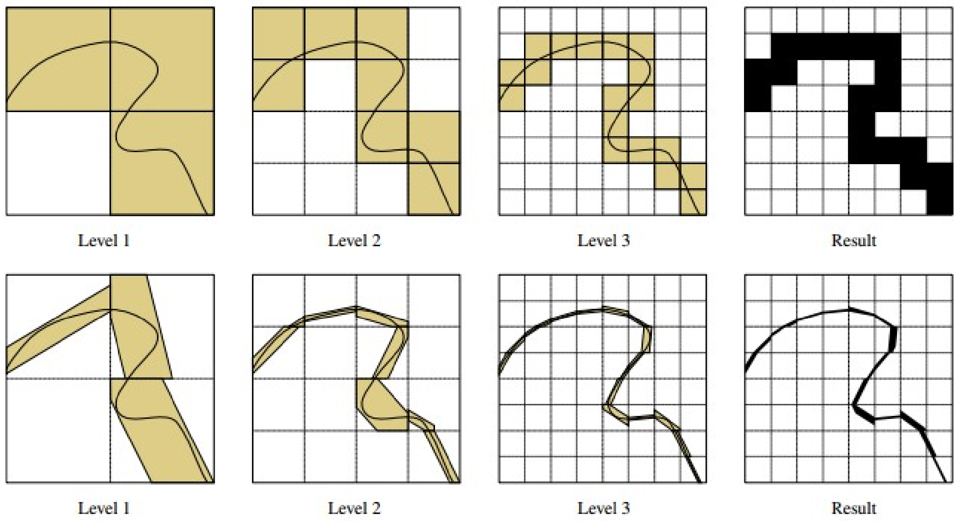


Figure A2: Effects of contours on surface approximations. [3]

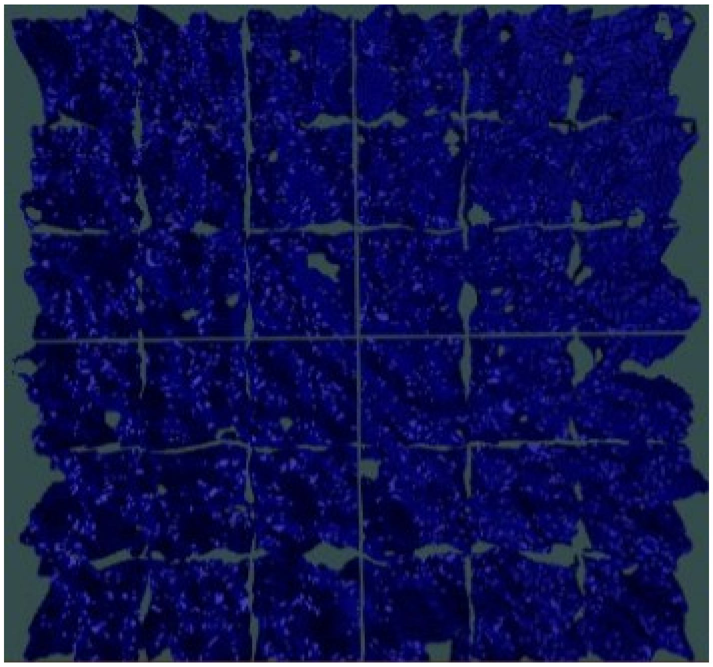


Figure A3: Seam between chunks of marching cube meshes.

8. Biography

Richard Dean Schaefer is completing his final year at the University of Florida pursuing an undergraduate degree in the College of Engineering for Computer Science, he plans on graduating in April of 2018. Richard has completed internships with Allstate Benefits in Jacksonville Florida and Immersed Games in Gainesville Florida. In past projects Richard has analyzed traffic data to predict accidents, programmed a 2D java platformer, and implemented the game logic of board game called *Carcassonne* along with an AI to play it. After graduation he plans on accepting a job offer with CSX in Jacksonville as a part of their software leadership development program. Richard has been passionate about programming since his first class in high school; since then he has become proficient in Java, Python and, increasingly, C++. Richard took on this project in an effort to hone his skills with the language and investigate the crossroads of his curiosity with manipulable terrain and 3D graphics. In his free time Richard enjoys playing games, a glass of good whiskey, and working on overly complicated C++ projects.