Bournemouth University

MSc CAVE - Masters Thesis - NCCA

Traversing Into Sparse Voxel Octrees

Giles Penfold

August 1, 2018

1 Abstract

Sparse Voxel Octrees are commonly used for efficiently storing voxelized data in both medical imaging and games. In this paper, we research into efficient ways to store these large datasets after they have been created. After researching into the history of voxelization, we then look into the applications of Sparse Voxel Octrees and attempt to further enhance the storage capabilities of binary voxels using common compression methods.

Key Words: Sparse Voxel Octree (SVO), Voxelization, Compression

Contents

2 Introduction

Voxelization has been an important foundation in volume computer graphics for a long time, providing a solid grounded method for volumetric calculations, rendering, and medical imaging over the past few decades [1] [2]. Voxels, deriving from the terms 'volume' and 'pixel', are often cubic representations of a volume in 3D space, similar to a pixel being a 2D representation of an area in 2D space. A voxelized scene can be quickly traversed, allowing fast access to spatial data, with voxelized objects granting faster execution of dynamic physics-based algorithms, such as destruction. Our focus will be on *surface voxelization*, which creates a voxel hull around the mesh and is more suited for rendering, compared to *solid voxelization*, which creates a full volume mesh.

Voxelization has come a long way since the scanline methods of the 80's, which were based upon 2D rasterization techniques. In the current decade, new methods have emerged within the field which allow for faster and more optimized calculations in voxelizing objects, as well as attempts to store this data efficiently. One such method was the Sparse Voxel Octree, proposed by S.Laine and T.Karras in 2010, which included compression methods for storing higher resolution meshes, smooth surfaces and high-precision normals [3]. This was taken further with the Out-Of-Core (OOCSVO) method by J.Baert in 2013, who proposed a novel way of producing SVOs using an alternative to the original In-Core method, which helped save on memory [4].

This research will look into the foundations of voxelization in Computer Graphics, and its applications. We will then investigate constructing our own In-Core SVOs and look into potential methods of optimizing further when storing the data they produce. Our method will focus around the use of binary voxels with the enhancement of techniques found in Baert's paper, as well as use of compression methods such as Miniz [5] and Snappy [6].

3 Voxelization

In this section, two common stages found in the voxelization process are looked at: Triangle Voxelization and Storage Grids. The first area covers how triangles are converted into voxels, with the second looking into how these voxels are stored in memory for efficient lookup and traversal. Within voxelization, there is *surface voxelization*, which only voxelizes the surface of a mesh, and *solid voxelization*, which voxelizes the surface and interior space of a mesh.

Four methods of triangle voxelization are commonly used in current implementations across various GPU packages[10]. The first method was suggested by Huang in 1998 and still holds true as one of the most reliable methods of triangular *surface voxelization* today, even stretching into quadrangles and other geometric shapes [7]. The second method is another *surface voxelization* process first suggested by Tomas Akenine-Mllser in 2001, which was later adapted by Schwarz in 2010 [9] [8]. This method is called Triangle-Box Overlap (TBO). The third and fourth methods, Triangle-Parallel Solid and Tile Based Solid, were suggested by Schwarz in the same paper [8] and are more commonly used in larger GPU accelerated implementations such as in NVIDIA's NVAPI [10], as well as in voxelized rendering pipelines [11]. Both these methods are *solid voxelization* algorithms.

Our implementation will focus on Schwarz's method of TBO as it is a robust *surface voxelization* method which allows for easier implementation on a GPU should we wish to parallelize, whilst still providing a fast and effective CPU based algorithm.

3.1 Voxelizing Triangles: Surface Methods

Geometrical

In Huang's paper, certain problems previously found in voxelization are addressed, such as dealing between infinite and finite planes. In section 5 of the paper, polygonal meshes are addressed and a solution is proposed to voxelizing these finite triangular planes. The method focuses on the border conditions of a triangle being mapped into a voxel, whilst maintaining reasonable separability and minimality. Separability is the continuation of data between local and world space, with minimality being the use of the smallest number of voxels possible to describe a triangular polygon [7].

Three sets of tests are constructed to determine the voxels in which a triangle lies. First, a bounding sphere of a set radius, r, is constructed on each vertex of the triangle, with any voxels lying within these spheres are added to the voxel list for this triangle. Next, a bounding cylinder of equal radius to the spheres, r, is constructed along each edge of the triangle. All voxels within these bounding cylinders are then added to the voxel list. Finally, two triangular bounding planes, G and H, are constructed parallel to the triangle, followed by additional planes along each edge, connecting G and H together to form a closed boundary. Any voxels within this closed boundary are then added to the voxel list [7]. This can be seen visually in Figure 1.

The final union of these three tests in the voxel list provide a voxelized representation of the triangle. This is repeated over a polygonal mesh with all lists being combined together to form a final voxelized mesh [7]. This method

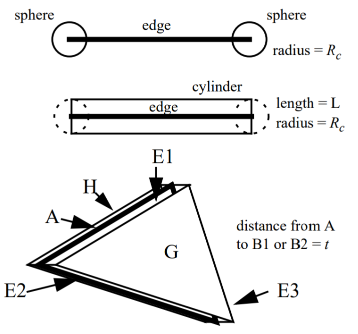


Figure 1: Visual representation of Huang's voxelization method. Credit: Jian Huang [7].

provides a simple method of voxelizing a triangular polygon, which, most importantly, follows the requirements of separability and minimality, unlike other methods before it.

Triangle-Box Overlap

Schwarz states that conservative voxelization should identify all triangle/voxel overlaps in a fast, simple test as well as supporting an effective setup stage that depends only on the triangles and the voxels [8]. Conservative voxelization is the process in which voxels are created around a surface so that no voxel only touches the diagonal edge of another voxel. A surface face of a voxel must be touching with the surface face of another voxel along the path of voxelization. In this method, given a triangle, T, and an axis-aligned voxel, V, that T overlaps V if: The plane of T overlaps V and each of the coordinate planes (xy, yz, xz), when projected from T onto V, also overlap [8].

Given the triangle has a bounding box at the start of the calculation, a large number of the preliminary values, such as the edges, the normal and a critical point, can all be calculated. A planar overlap test is first performed, using the critical point to determine if the plane lies of different sides of the voxel. Edge functions are then utilized to perform the 2D projection overlap tests. A normal and dot product value are calculated for each edge and tested for overlap. Each edge must have a separate calculation performed as each of the evaluation points of the edge tests differ. Each of these ten tests, the planar overlap and the three planes upon the three edge overlaps, must pass for the triangle to be accepted as intersecting with the voxel in a conservative manner [8].

This method proves a robust and fast algorithm for *surface voxelization* which could be adapted to a GPU version in the future within our implementation. It also supports both In-Core and Out-Of-Core functionality [4].

3.2 Voxelizing Triangles: Solid Methods

Triangle-Parallel Solid

The first of the solid methods described by Schwarz, Triangle-Parallel Solid(TPS), is a GPU centric approach that uses planar projections to achieve voxelization [8]. A bounding box is projected into the yz plane from a triangle, with any intersecting voxels indicated as candidates. If candidates are found, they are looped over column-wise and tested against the yz projection using edge functions. If this test succeeds, the voxels binaries are flipped, indicating a state change from empty to full. Any overlaps between triangular projections are shared, with successive voxels not being flipped again to avoid conflict between triangles.

TPS can be parallelized easily on the GPU by splitting the triangles into separate buffers and processing these buffers in parallel [8]. This method is not the most optimal for this implementation, as it revolves around GPU optimization, where as the focus is on a CPU implementation. TPS also involves a greater amount of processing time compared to TBO as it is a *solid voxelization* method.

Tile Based Solid

The second method described by Schwarz, Tile Based Solid(TBS), is another GPU centric approach which attempts to sidestep the problems with triangle based methods in parallelization: the number of voxel columns varying drastically between processing each triangle, and the flipping on voxels in a large mesh can result in a massive memory overhead, which can cause the system to crash.

First, tiles are assigned along the yz plane in a collection of 4x4 voxel columns. One thread can then be dedicated to each tile, which act independently of each other and do not run the risk of large memory overheads. For each triangle, and for every parallel triangle, it is determined which tile overlaps them as they are then split into buffers. A work queue is predicted, deciding how much overhead each tile contains and triangle pairs are formed within tiles for further parallelization. A triangle can exist across multiple tiles, and as such a bounding box is created around the triangle to determine which tiles it falls within and then using edge functions to double check the tile allocations. Schwarz states this is more computationally expensive than just allocating tiles by bounding box, but instead shortens the work queue, reduces memory overhead and speeds up the sorting process of triangles later on [8].

Once triangles have been allocated, the TBS algorithm then processes each triangle in a tile sequentially, processing multiple tiles in parallel. Each voxel column is assigned a thread and each thread contains its own segment to avoid large global memory overheads. Each triangle is then processed and voxels are flipped.

This method is much more complex than the previous method, and heavily relies on parallelization on the GPU to remain efficient. While it is an exceptionally fast method, our implementation does not adopt this due to time constrains and complexity.

3.3 Grids

The voxelization process of the triangle is only half the work required over a complete implementation. Usually, interleaved with this process, is a method of storing the voxels so that they do not create a large memory overhead and so that they can be accessed quickly in real-time. The process for doing this normally involves the use of grids.

Scanline

One of the oldest forms of voxel storage involves a scanline approach similar to 2D rasterization techniques. A large grid would be placed over the mesh and divided up into cubes, which would represent a voxel. A method of voxelization would be used to on the grid, such as Huang's Geometrical method [7], which would start at the top and work across, moving along the x axis, then the y axis. When a voxel is created, the algorithm will then search across the grid in the z direction for an adjacent voxel, checking each cell for a triangle. Once one is found, it fills in all the voxels between the two to create a *solid voxelized* mesh. It then returns to the original point and repeats this process until the entire volume of the mesh has been filled [12]. This can be used against for lookup of a voxel, giving the index of the row and column of the grid and scanning across to find said voxel.

This method has existed for over 3 decades, and continues to be an effective method for creating solid voxel meshes within a CPU based implementation. It is, however, slower than TPS, which can handle large sections of voxels at once. There have been adaptations of scanline before TPS which follow a similar trail of though, such as a Shear-Warp algorithm [13] , which has been successfully parallelized [14].

Sparse Voxel Octrees

Octrees have been a good method for data storage since their creation [15], used for collision meshes [16], colour quantisation [17], and voxelization [3]. In Laine and Karras' paper in 2010, a sparse method was suggested in using octrees to represent voxel data, which was geared towards optimizing rendering of complex scenes. As this revolved around optimizing for rendering, methods for normal compression, smooth surface encoding and ray cast acceleration were included in this paper, which was a large step forwards for voxelizations potential in GPU rendering [3].

Whilst being a large step forwards, these methods were often limited as being in-core memory algorithms which were constrained to the size of the RAM on the target machine. Baert in 2013 proposed an out-of-core method which allowed for reading and writing from external memory drives whilst performing this voxelization [4].

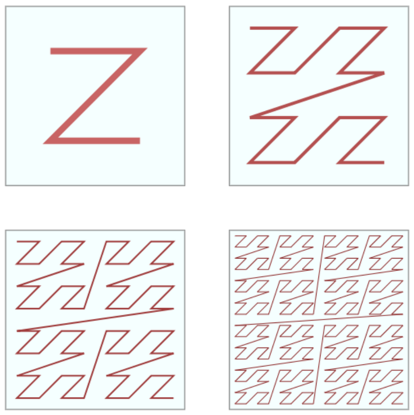


Figure 2: Different tiers of a 2D Morton order grid. Credit: David Eppstein

Out-Of-Core Sparse Voxel Octrees

Baert proposes a method of partitioning using morton order subgrid construction of a mesh before computation, which allows for partitioning of the mesh. This data can then be stored to file as separate partitions and worked upon independently before being written to file in an octree file format. The process heavily revolves around maintaining a morton order throughout the subgrids, partitions and octrees. This is so the algorithm can easily track the start and end points of each partition with ease, and iterate quickly through each voxel within each partition in order [4].

The main algorithm starts by constructing a morton order subgrid around the mesh, conforming to a power of 2. Once this subgrid has been constructed, the triangles of the mesh are allocated to grids and stored under partitions depending on the available memory. The algorithm will then save each partition to file and work on each independently in the memory allowance, writing each finalized partition out after processing has been completed. A final octree file is then constructed from this, which will be large enough to be read into the allocated memory for that computer.

The final octree is constructed using a number of build queues, which are arrays of 8 binary nodes, with a binary node representing the on/off state of a voxel and references to its children. These queues are stacked in a hierarchical way, with the lowest queue being attributed to the first node of the previous queue. The lowest queue is filled to a maximum size of 8, and then all the non-empty nodes are referenced up to the first node of the parent queue. This queue is then cleared and the process repeats for the second node of the parent queue. Once the parent queue is full, it undergoes the same process with the queue above it, forming a tiered storage system for the octree.

This is the method that shall be used for this implementation of SVOs, using Baert's source code as a reference point and further looking into possible compression methods for the final octree file. As the implementation is focusing on the compression of the final octree file, an In-Core method shall be used instead of an Out-Of-Core, which allows for simpler testing of compatibility with various compression methods.

4 Implementation: Sparse Voxel Octrees

This section looks at the construction of SVOs and the compression of the octree data used for the storage of these volumetric objects. First the organization of the mesh before voxelization shall be looked at, using Morton ordered grids to create a quickly traversable sub-grid over the mesh. Next Schwarz voxelization of the mesh is implemented, followed by the organization of the voxel data into an octree which follows the Morton order of the subgrids. Finally, a handful of compression methods are tested on the octree data and compared against the original, uncompressed data.

4.1 Morton Grids

Morton ordered grids are the first step in a fast voxelization method, especially if the process is to be taken OOC. A Morton order, or Z order, grid follows a specific pattern. In a 2D implementation, a square can be divided into 4

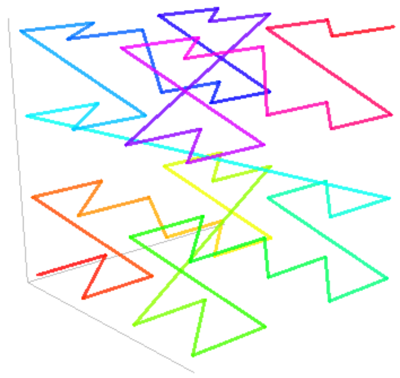


Figure 3: A 3D Morton order grid. Credit: Robert Dickau

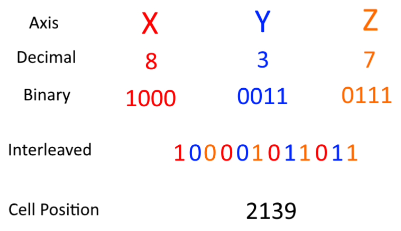
sub-squares, labelled 1, 2, 3 and 4 following a Z curve as seen in the top left example of Figure 2. This can then be iterated on for each sub-square, dividing each of these into concurrent sub-squares and labelling off in the same manner as before, starting at the lowest level of the tree structure and moving across as each sub-square is indexed.

Figure 2 shows the second level of Morton order in the top right, the third level in the bottom left and fourth level in the bottom right in a 2D scope. This can, however, be extrapolated to the 3D domain following the same principles as in the 2D domain. A low level representation of this is displayed in Figure 3.

Morton Encoding

Morton encoding stores the index of a cube in 3D space along a predefined grid as a 32bit integer [18]. The bits of each axis value: x, y and z are interleaved in a certain pattern so that they can be used to reference a certain cell on the grid. The lower 16 bits of x and y are interleaved so that the x bits are stored into the even positions and the y bits are stored into the odd positions [18]. These are then interleaved into z to form a 32bit integer which gives the cell position along the grid.

Baert converted this method to work with 64bit integers and involves looking at 21 bits of each axis value before interleaving them into a 64 bit integer. Five shift operations are performed on each of the integer values which then place them into the correctly interleaved bit positions on the 64bit integer. This can also be reversed, given a Morton 64bit integer, the bits can be shifted back to attain the grid co-ordinates in x, y and z terms [19]. A basic 12bit example can be seen in Figure 4.



Axis	X	Y	Z
Decimal	8	3	7
Binary	1000	0011	0111
Interleaved	100001011011		
Cell Position	2139		

Figure 4: Morton Encoding

The Sub-Grid Process

As the mesh is loaded into the scene, the NGL library is used to calculate the bounding box of the mesh, attaining an AABB bounding box. As the Morton grid requires a power of 2 to correctly sub-divide, a calculation is performed on the bounding box to create a cube instead of a rectangle. This calculation takes the height, width and depth of the original bounding box and finds which of the three is the largest. It then conforms the other two lengths to expand so that the height, width and depth are all the same and safely fit around the mesh. This way, no section of the mesh is cut out. Each of the vertices of the mesh are then translated in relation to the origin so that the mesh is central to the scene.

Next, memory calculations are performed on the projected size of the grid. The number of partitions that will be needed are calculated during this step. This pertains to the OOC method where partitions can be stored to disk between operations, and also allows to faster segmentation and traversal of the tree structure [4]. In the standard In-Core method, only one partition is ever used [3]. These partitions will also follow Morton order and conform to powers of 2. It is possible to calculate the number of Morton cubes within a partition by cubing the total size of the grid and dividing by the number of partitions. This gives us a Morton encoded 64bit integer. This value can be decoded and used to allocate a bounding box to the current partition. This is performed for each of the partitions.

Each triangle in the mesh is then allocated a bounding box within NGL and tested against the bounding box of each partition. If it exists within a partition, it is allocated to that partitions triangle list. Once all triangles within the mesh have been allocated to the Morton grid, the voxelization can begin.

Algorithm 1: Morton Construction for Voxelization of a Mesh

```
Data: Triangle List TL, Grid Size G, Memory Limit M
Result: Morton Grid MG
Calculate Number of Partitions N using G and M;
Calculate the Morton partition value MP using G and N;
foreach Partition P out of a total N do
    Decode Morton of P in relation to MP;
    Attain the Min and Max values of the bounding box for P;
    Construct the bounding box of P;
    Add P to MG;
end
foreach Triangle T in TL do
    Calculate bounding box of T;
    foreach Partition P in MG do
        if T is inside of P then
            Add to the triangle list of P;
        end
    end
end
```

4.2 Voxelization

After the construction of a Morton ordered grid that contains all the triangles of the given mesh, voxelization can finally be performed. As seen in Section 3, the Schwarz TBO method of voxelization proved to be the most robust in modern CPU methods given the *surface voxelization* that it provides [8].

Triangle-Box Overlap

First, each partition is iterated over and checked for triangles. If no triangles are found, then the partition is discarded, as attempting to voxelize empty grid spaces is highly inefficient. Assuming triangles do exist within the partition, however, they are then iterated over. As each triangle already has a bounding box which exists within the world space, it is first clamped to the grid space so that it can be checked against the individual parts of the Morton grid. If the triangle was left in world coordinates, then the tests would not be fully accurate as the triangle might seem to appear outside of a sub-grid it was actually inside of. This is the *separability* that was talked about in Section 3.1 [7].

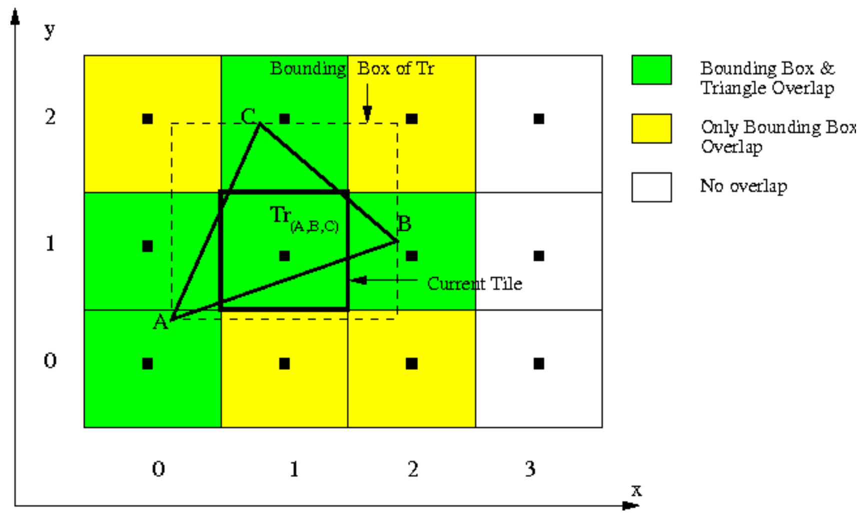


Figure 5: A simplified conservative triangle/box overlap test. Credit: Iosif Antochi

A series of properties are extracted from the triangle, which includes each of the 3 edges and the normal of the triangle surface. A critical value is also extracted, which determines the direction the triangle's normal is facing and makes it uniform to the unit length of the grid space [8].

Moller states that the first test should be for a bounding box overlap, however, this was already done in the construction of the Morton Grid, so the second set of tests can be skipped to. A planar overlap test is constructed first, as described in Section 3.1. This tests the two diagonal vertices whose direction is most closely aligned to the normal of the triangle [9]. Edge tests are created for each of the 2D projections along the edges of the triangle (xy, yz, xz). This involves creating a normal and dot product calculation for each edge along each projection [8]. There are 9 of these edge tests in total, 3 for each edge of the triangle. Where $i, j \in \{0, 1, 2\}$, and $\mathbf{e}$ is the normal of an edge for $\mathbf{a}_{ij} = \mathbf{e}_i \times \mathbf{f}_j$ where $\mathbf{f}_0 = \mathbf{v}_1 - \mathbf{v}_0, \mathbf{f}_1 = \mathbf{v}_2 - \mathbf{v}_1$, and $\mathbf{f}_2 = \mathbf{v}_0 - \mathbf{v}_1$ and if $\mathbf{a}_{ij} < 0$ the test passes [9].

After these tests have been created, they are then resolved against each voxel within the projected sub-grid that the triangle's bounding box occupies. First the plane test is performed, using dot product values and the normal of the triangle. This checks if the plane intersects through the voxel box. Following this, each of the edge projections are tested against the voxel. If all of the ten tests pass, this voxel is allocated as filled. These tests ensure that *minimality* and *conservative voxelization*, which were talked about in Section

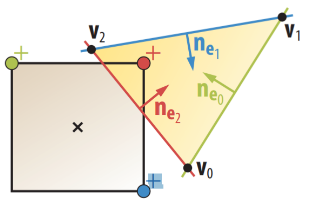


Figure 6: Critical points of the triangle for conservative voxelization. Credit: Michael Schwarz [8]

3.1, are maintained [7][8].

This process repeats for each triangle within every partition. Multiple triangles can exist over different partitions, so there is some overlap. This means that the algorithm is fully enclosed and leaves no holes in the final voxelized surface, providing a conservative voxelized mesh.

4.3 Sparse Voxel Octree Construction

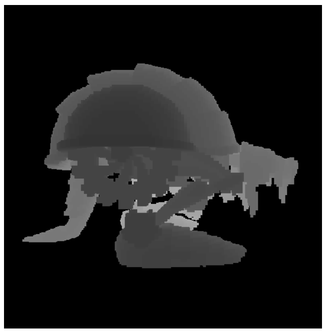
In this section, the creation of a SVO is discussed, which uses the voxelized 3D morton grid from the previous section as an input. A fully structured SVO is expected as the output of this process, which should have good locality where sibling leaf nodes are stored near each other.

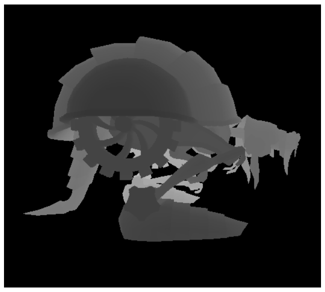
Construction

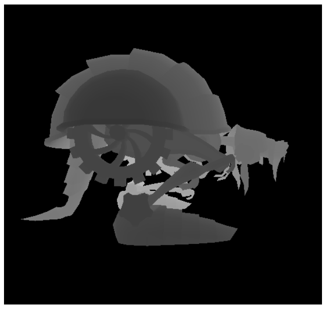
To begin, an empty octree, **O**, is constructed and the current morton of this octree is allocated to to 0. The maximum depth, **D**, of the **O** is the natural logarithm of the size of the morton grid to the base 2. Memory is reserved for each of the queues, known as child arrays, giving each a length **D**. These will accept the voxel data using Baert's queueing method [4]. Compression flags are then set for the algorithm to track what type of compression to use, and a queue is setup to track when to compress the files.

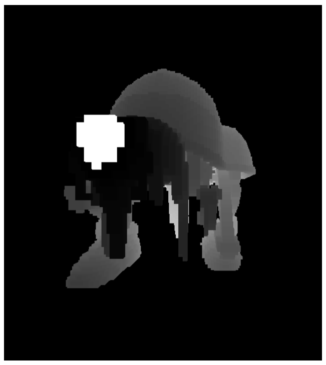
Each of the child arrays holds a node, which contains: Data about the node (filled or empty), the base morton position to search from to find the children of the node, an array of characters equal to length **D** to indicate the state of the children of this node. In this array, -1 indicates an empty node and any positive integer <= **D** indicates the number of filled nodes in that child node.

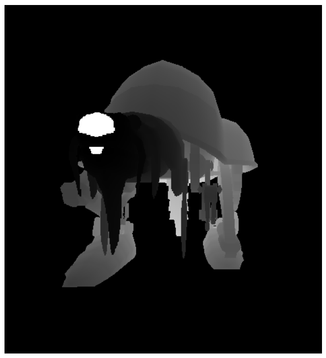
The partitions of the morton grid are then iterated through, adding voxels to **O**, which starts with the lowest child array, **LCA**, in the list. Once **LCA** is full, a new node, **n**, is created and allocated the information of **LCA**. It is


(a) A model voxelized with a 128 size morton grid, side view.


(b) A model voxelized with a 1024 size morton grid, side view.


(c) A model voxelized with a 4096 size morton grid, side view.


(d) A model voxelized with a 128 size morton grid, front view.


(e) A model voxelized with a 1024 size morton grid, front view.

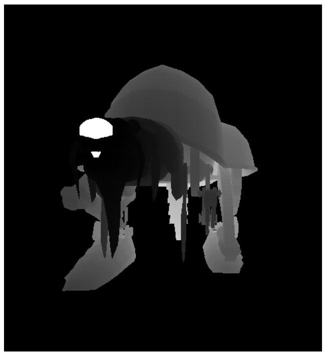

(f) A model voxelized with a 4096 size morton grid, front view.

Figure 7: Renders of the final voxelization process

then placed into the child array one depth level above **LCA**. This continues until the entire octree has been filled. If any empty voxels are encountered, the number of nodes that should be filled with empty voxels (and their children) is calculated, before quickly being filled and allocated up the child arrays. This can be seen in Algorithm 2.

The final octree visulatization can be seen in Figure 7, which shows the selected model rendered with a basic voxel renderer. When rendered, anything above a 1024 grid size has reasonably negligible changes in the visuals, but if we were to process volumetric algorithms through it, or attempt to render volume using the octree, it would certainly have more of an effect.

It was found that anything above a 1024 grid also caused memory overflow on limited memory resources, often going over the desired 8GB limit. Further optimizations could be implemented in the future to combat this using an Out-Of-Core method.

The data is being compressed as voxelization occurs, so as each node is written into an array, it is also added to a compression queue. This queue will cap at 1000, or any arbitrary limit decided that after which it reaches it shall write to disk. The longer the length between writes to disk, the more likely errors are to occur in the compression and with any potential corruption to the file. This is because the compression algorithms often use carriage returns of common operating systems and white space as symbols within the compression, which can cause issues when transferred across systems.

This problem is combated by using delimeters after the compression has been performed, but before it has been written to disk. Any **\r** and **\n** symbols that are legitimate compression symbols and not carriage returns are checked for. A delimeter is then wrapped around the symbol, so that if it is lost within the read/write process, it can be recovered at a later date.

In the event that an erroneous value is allocated, the situation can be resolved by making an intelligent guess at what the system might need. In the case of the data of the nodes, it can be set to empty if the surrounding nodes are empty, or filled if they are filled. As for the child offset and arrays, this is harder to determine. Currently, the data is set to 0, which may result in data loss of an entire branch of the octree, but continues with the algorithm correctly after breaking from that branch. This is the most optimal way of dealing with the potential issue, but not necessarily the most effective in terms of data preservation. Given the delimeters are padded out to be a specific set of characters of length 6 and the characters used contain 64 to 512 potential candidates, the chances of an error with a carriage return is 64^{-6} to 512^{-6} or between 1 in a quintillion to 1 in a septillion. While these chances are relatively low, they are still feasibly possible within large data sets. Improvements on this would be increasing the delimeter length, which will decrease chances of errors largely but also increase the compressed file size. Increasing to length 10 would decrease chances to one in a nonillion (10^{-30}) and size 20 to one in a novemdecillion (10^{-60}).

Algorithm 2: Sparse Voxel Octree Construction

```
Data: Ordered Morton Grid OMG, Empty Octree O
Result: Sparse Voxel Octree SVO
Set the maximum depth D of O;
Allocate maximum number of children of O to 8 * D;
Setup the compression queue;
foreach Partition P in OMG do
    foreach Morton M in P do
        if Leaf Node Morton LM of the Lowest Child Array LCA of O
          contains empty nodes then
            Compute largest a where $8^a <$ number of empty nodes;
            Compute largest b where child array[b] of O is not empty;
            Compute $d_i = \max(D-a, b)$;
            Add empty nodes to child array[$d_i$];
        end
        if LCA is not full AND D > 0 then
            Add M to LCA;
        else
            Create a new internal Node N;
            Add LCA as the child offset of N;
            Clear LCA;
            Add N to the compression queue;
            Add N to the child array of O that is LCA-1;
            D–;
        end
        if Length of the compression queue > 1000 then
            Compress each node in the compression queue together;
            Write the queue to disk;
            Clear the queue;
        end
    end
end
```

4.4 Compression & Results

In this section, different compression techniques that were tested with the octree dataset are discussed. Two forms of compression were implemented into the program that can inherently read and write data from a custom ".octreecompress" file. Other external methods were also tested by compressing the original ".octree" file generated by the program.

Each of these methods were tested on a 2.47MB obj file, which produced a 96.5MB .octree file of grid size 1024. This was tested on a Intel i7-5700HQ at 2.70GHz with 4 cores and 8 logical cores, reading/writing to a Intel 360GB SSD. 8GB RAM was allocated as the maximum allowance for the program.

In these tests, 100 iterations over each method were performed and the average time for the voxelization was taken. Compression and writing to disk were timed separately. The files are then loaded back in and 100 more tests are made before taking the average read time. With no compression taking place, the average processing time was 23.04 seconds, with a write time of 1.02 seconds and a read time of 1.24 seconds.

Snappy

The first compression method within the program is Snappy, which is a light data compression library created by Google to assist in the compression of their Remote Procedure Call systems as well as in numerous open source database systems. [6] The library was originally created with speed in mind, not maximising compression size, giving very fast read and write times. This method was chosen for the fast read/write times to be compared with the other methods.

To process the Snappy compressed octree, the average voxelization time resulted in 48.03 seconds, over 2x longer than with no compression. The average compression/write time was 79.28 seconds, with a decompression/read time of 82.64 seconds. Despite this, the compressed size of the final file was 13.3MB, 7.25x smaller than the original file.

Miniz

The second compression method used within the program is Miniz, which is a single header C library based on the zlib library. This method supports the writing of zip archives, as well as dense compression whilst maintaining faster speeds than zlib [5]. This library was chosen for the good compression it gives as well as the faster processing time compared to larger libraries.

To process the Miniz compressed octree, the average voxelization time resulted in 49.70 seconds, only slightly longer than Snappy. The compression/write time was 1.282 seconds and the decompression/read time was 1.43 seconds, which correlated more towards the original read time. Miniz also produced the best file size of the three options, being 3.62MB, 26.65x smaller than the original file. Despite having a longer processing time, Miniz provided the most effective method of compression out of the two implemented into the software, giving both smaller file sizes and faster read/write times.

Other

A number of other compression methods were tested externally to the program. Zip is a common compression type found across most operating systems, with Rar being a strong compression method commonly found on Windows. Despite being encoding methods instead of compression, Base32 and Base64 have been known to compress certain datasets and such were tested for completeness. These two methods were implemented into the program but later removed due to poor results. The external ".octree" file written by the program was compressed using each of these systems and then uncompressed, compared to the ".octreecompress" files which have independent data written into them. It should be noted that Zip performed well, compressing the file in under 1.42 seconds and decompressing it in 1.539 seconds, only slightly slower than Miniz. It produced a file size of 4.43MB, which was also larger than the Miniz file. Rar compression took 2.452 seconds to compress and 2.511 seconds to decompress the file, giving a total file size of 3.88MB. This was done using the "Best" compression method Rar provides in its service.

Base32 and Base64 were tested early on in the project, despite not being compression methods. They provided average write times of 1.830 seconds and read times of 2.03 seconds, but the file sizes were larger than the originals. Base32 provided a 106MB file and Base64 provided a 109MB file.

5 Conclusion & Future Work

Our look into Sparse Voxel Octrees proved beneficial to understanding the core concepts in the compression of volumetric data. These octree structures are far more complex than the basic octree structures used in collision detection and simple rendering, providing methods for storing as much information as possible into the nodes without overburdening the memory or storage capacities. Only binary node data was being dealt with, this data was easily compressible by conventional methods and proved possible using current compression libraries and software.

While the code worked successfully, further optimizations could be made in the read/writing of the files. This would potentially involve the removal of the data clustering during compression, which would allow for faster read/write times, however the downsides would be the lack of error correction after the decompression has taken place. In the event that the file had become corrupted or changed, the system would not be able to successfully load in the data and make attempts to correct it.

It would have been good to involve visualization of the octrees in the program itself, however this proved too heavy with initial trials and writing a full voxel renderer was beyond the scope of this project. It would also be of interest to follow further into Baert's system of Out-Of-Core octrees and to implement both In-Core and Out-Of-Core methods of processing the octrees, rather than just In-Core, so that the system can handle exceptionally large datasets and larger grid sizes.

Overall, the project was a success, giving an insight into the processes of voxelizing triangular models, storing the data of this efficiently and then compressing that data for effective storage solutions.

6 Acknowledgements

Thanks is given to Jon Macey and Dr. Ian Stephenson for their helpful input during the course of this thesis, and helping to push ideas in different direction.

I would also like to thank my coursemates on the MSc CAVE course, my friends, girlfriend, and most importantly, my parents for constantly providing support and advice throughout my year at Bournemouth.

Bibliography

[1] Hutton C, et al. *Voxel-based cortical thickness measurements in MRI* Neuroimage. 40(4): 17011710. May 1, 2008.

[2] Karabassi E-A, Papaioannou G, and Theoharis, T. *A Fast Depth-Buffer-Based Voxelization Algorithm* Journal of Graphics Tools Volume 4, Issue 4, Pages 5-10. 1999.

[3] Laine S, and Karras T. *Efficient Sparse Voxel Octrees.* Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games. Pages 55-63. 2010.

[4] Baert J, et al. *Out-of-Core Construction of Sparse Voxel Octrees.* In Proceedings of the 5th High-Performance Graphics conference, pp. 27-32. ACM, 2013.

[5] Geldreich R. *Miniz* https://github.com/richgel999/miniz

[6] Google *Snappy* https://google.github.io/snappy/

[7] Huang J, et al. *An Accurate Method for Voxelizing Polygon Meshes.* IEEE Symposium on Volume Visualization. 119-126. 1998.

[8] Schwarz M, and Seidel, H. *Fast Parallel Surface and Solid Voxelization on GPUs.* ACM Transactions on Graphics (TOG) 29, no. 6 (2010): 179. 2010.

[9] Akenine-Mllser T. *Fast 3D triangle-box overlap testing.* Journal of graphics tools. 2001

[10] NVIDIA *NVAPI* https://docs.nvidia.com/gameworks/content/gameworkslibrary/coresdk/nva

[11] NVIDIA *GVDB* https://developer.nvidia.com/gvdb

[12] Pantaleoni J. *VoxelPipe: A Programmable Pipeline for 3D Voxelization* In Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, pp. 99-106. ACM, 2011.

[13] Lacroute P, and Levoy M. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation* Computer Graphics Proceedings, SIGGRAPH. 1994.

[14] Amin M B, et al. *Fast Volume Rendering Using an Efficient, Scalable Parallel Formulation of the Shear-Warp Algorithm* In Proceedings of the IEEE symposium on Parallel rendering, pp. 7-14. ACM, 1995.

[15] Donald M. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer* Rensselaer Polytechnic Institute (Technical Report IPL-TR-80-111). 1980.

[16] Leubke D P. *Level of Detail for 3D Graphics* Morgan Kaufmann. ISBN 978-1-55860-838-2. 2003.

[17] Gervautz M, and Purgathofer W. *A simple method for color quantization: Octree quantization* In New trends in computer graphics, pp. 219-231. Springer, Berlin, Heidelberg, 1988.

[18] Anderson S E. *Bit Twiddling Hacks* http://graphics.stanford.edu/s̃eander/bithacks.html#InterleaveBMN

[19] Baert J, et al. *Morton encoding/decoding through bit interleaving: Implementations* http://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/