FAST GPU GENERATION OF SIGNED DISTANCE FIELDS FROM A VOXEL GRID

by

NICOLAS FEDOR
URN: 6683787

A dissertation submitted in partial fulfilment of the
requirements for the award of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

May 2025

Department of Computer Science
University of Surrey
Guildford GU2 7XH

Supervised by: Joey Sik Chun Lam

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

Nicolas Fedor
May 2025

Abstract

Write a summary of the work presented in your dissertation. Introduce the topic and highlight your main contributions and results. The abstract should be comprehensible on its own, and should no contain any references. As far as possible, limit the use of jargon and abbreviations, to make the abstract readable by non-specialists in your area. Do not exceed 300 words.

Acknowledgements

Write any personal words of thanks here. Typically, this space is used to thank your supervisor for their guidance, as well as anyone else who has supported the completion of this dissertation, for example by discussing results and their interpretation or reviewing write ups. It is also usual to acknowledge any financial support received in relation to this work.

Contents

List of Figures

List of Tables

Glossary

P	Placeholder

Abbreviations

SVO	Sparse Voxel Octree
SVDAG	Sparse Voxel Directed Acyclic Graph
DDA	Digital Differential Analysis
LUT	Look-Up Table
LOD	Level of Detail
RLE	Run-Length Encoding
SFC	Space Filling Curve
FPS	Frames Per Second
BVH	Bounding Volume Hierarchy

Chapter 1

Introduction

1.1 Problem Statement

Rendering large-scale voxel worlds with dynamic scenes presents significant challenges due to the vast amount of data required to to represent 3D environments. Voxel-based rendering, where the scene is composed of volumetric pixels (voxels), is an attractive alternative to traditional triangle-based rendering for its ability to represent complex geometry and volumetric effects; however, the scalability of voxel-based rendering is limited by the amount of memory required to store the voxel data, and often times the computational cost of updating hyper-compressed voxel data structures.

Sparse Voxel Octrees (SVOs) (Laine & Karras 2010) are a popular data structure for representing voxel scenes, as they provide a compact representation of the scene by storing only the occupied voxels in a tree structure. Octrees, or nodes, can be dynamically subdivided to increase the resolution of the scene, and provide a straighforward way of introducting Level of Detail (LOD). Octrees are also well suited to ray tracing, as they provide a natural way of separating a scene into bounding boxes that can be quickly tested for intersection (Ize 2009). SVOs allow for high compression ratios, and performant ray tracing, for the rendering of large (typically in the millions) of voxels.

SVOs are not without their limitations, however. Updating an SVO can be a straightforward process as only the affected nodes need updating; however, an SVO is usually further compressed into a format better suited for GPU rendering, using techniques such as run-length encoding (RLE) (Eisenwave n.d.*a*) and space filling curves (SFC) (Eisenwave n.d.*b*). Updating

these compressed formats, and transferring the new data to the GPU, can be computationally expensive (Crassin 2012). This makes SVOs less suitable for dynamic scenes where voxels can be added, removed, or modified frequently.

1.2 Aims and Objectives

The primary goal of this dissertation is to investigate the feasibility, and identify potential techniques and approaches that could improve the performance of using SVOs for rendering large-scale dynamic voxel scenes. The aims of this dissertation can be broken down into the following objectives:

1. Investigate current techniques for constructing and rendering SVOs.
2. Design a system for updating SVOs, on the GPU, to allow for large dynamic voxel scenes.
3. Develop a renderer, using Vulkan, that addresses the challenges of data transfer between the host and device of large SVOs.
4. Evaluate the performance of the system, and identify potential areas for improvement. See Section 1.2.1.
5. Investigate potential techniques for further improving the performance of SVOs for dynamic scenes.

1.2.1 Performance Metrics

The performance of the system will be evaluated using the following metrics:

Frames Per Second (FPS) The number of frames rendered per second. At a minimum for real-time rendering, and for interactive applications such as games, the system should be able to render at a consistent 30 FPS.

Since FPS isn't always a good indicator of performance, frame time should also be considered and is measured as the time taken to render a single frame including the update and render time. For a consistent 30 FPS, the frame time should be an average of 33.33ms.

Memory Usage The amount of memory used by the SVO on the GPU. The amount of memory an SVO uses is heavily dependant on the resolution of the scene, and the exact compression techniques used. For a 512^3 voxel scene, with a 9 level SVO, the memory usage should be less than 1GB (Crassin 2012, Laine & Karras 2010).

Data Transfer Time The time taken to transfer the updated SVO data between the host and device. TODO

Construction and Update Time The time taken to update the SVO on the GPU. Using the same 512^3 voxel scene, the construction time should be less than 100ms (Crassin 2012, Laine & Karras 2010); the construction of an SVO includes updating the tree structure with new voxels.

Rendered Voxels The number of voxels rendered in the scene. A 512^3 voxel scene consits of 134,217,728 voxels; having a fully populated octree would not be feasible and would mean a majority of the voxels are obsured. Using a suitable voxel scene, at least 1,000,000 visible voxels should be rendered at the 30 FPS target.

1.3 Scope and Limitations

A fully-featured voxel renderer is out of scope for this dissertation, this includes features such as:

Lighting and shading effects Effects such as ambient occlusion, refractions, reflections, and global illumination are not considered in this dissertation as they are renderer specific and not the focus of this dissertation.

Transparency and volumetric effects Rendering transparent voxels, or volumetric effects such as fog, smoke, or fire, add additional complexity to the rendering process, which is not the focus of this dissertation.

Identifying new techniques for SVO compression The focus of this dissertation is on the performance of updating SVOs, not on the compression techniques used to store the SVO on the GPU.

Optimizing the rendering engine A simple Vulkan renderer will be built to demonstrate the performance of the SVO update system, but the focus will be on the SVO update system itself.

1.4 Thesis Outline

1. **Introduction** - A brief introduction to the problem statement, aims and objectives, and the scope and limitations of the dissertation.
2. **Literature Review** - A review and discussion of the current techniques used for rendering voxel scenes, with a focus on Sparse Voxel Octrees; and the history of voxel rendering.
3. **System Design** - A detailed explanation of the system design, including the data structures used, the algorithms for updating the SVO, and the renderer design.
4. **Development** - A more in-depth look at the implementation of the system, as set out in Section 3.
5. **Testing and Evaluation** - A discussion on the performance of the system, and the results of the evaluation against the metrics set out in Section 1.2.1.
6. **Conclusion** - A summary of the dissertation, including the findings, limitations, and potential future work.

Chapter 2

Literature Review

Voxel rendering, and the associated techniques, have been an active area of research for many years. This chapter will provide an overview of the current techniques used for rendering voxel scenes, with a focus on Sparse Voxel Octrees (SVOs), and the history of voxel rendering.

2.1 Polygon-based Rendering

Traditional renderers use triangles to represent 3D objects, and the rendering process involves rasterizing these triangles to produce the final image. GPUs, and their graphics pipeline, are optimized for rendering large numbers of triangles.

Voxel grids are a simple way to represent a 3D volume of voxels, Minecraft is a popular example of a game that uses voxel grids for rendering as can be seen in Figure 2.1. However, voxel grids need to be converted into a series of triangles, and their associated vertex data, before the GPU can render them. This is a computationally expensive process, and can be slow for large voxel scenes.

There are several techniques to produce a polygon mesh from a voxel grid, two popular techniques are Greedy Meshing (see Section 2.1.1) and Marching Cubes.

2.1.1 Greedy Meshing

The naive approach to producing a mesh from a voxel grid is to iterate over each voxel, and for each voxel that isn't empty, create a quad consisting of two triangles for each face that is

Figure 2.1: Screenshot from the game Minecraft, showing a voxel world rendered using polygon-based rendering. Source: Minecraft Wiki

adjacent to an empty voxel. This approach is simple to implement, but produces a large number of triangles. Greedy meshing is an optimization to this approach that reduces the number of triangles produced by merging identical adjacent voxels into a single quad. The two approaches can be seen in Figure 2.2. Reducing the number of triangles can improve rendering performance as the GPU will have fewer triangles to rasterize; similarly there will be less vertex data that needs to be transferred to the GPU.

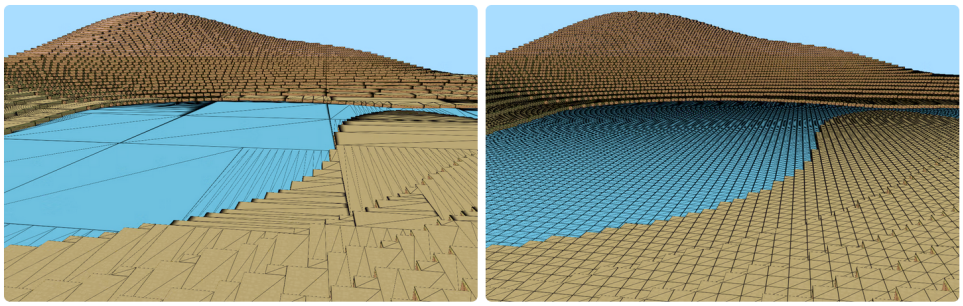

Figure 2.2: Comparison of a voxel grid meshed using the naive approach (right) and greedy meshing (left). Source: Jason Gedge (Gedge 2014)

2.2 Ray Tracing

Ray tracing is a rendering technique that simulates the way light interacts with objects in a scene. Rays are cast from the camera, and for each pixel, the ray is traced through the scene to determine the color of the pixel (see Figure 2.3). Ray tracing is well suited to voxel rendering due to the volumetric nature of voxels, and other optimisations such as bounding volume hierarchies (see Section 2.2.2 and Section 2.3). There are other advantages to using ray tracing for voxel rendering, such as the ability to directly operate on the voxel data providing more flexibility in the rendering process, instead of having to go through the intermediate step of converting voxels to triangles for rasterization.

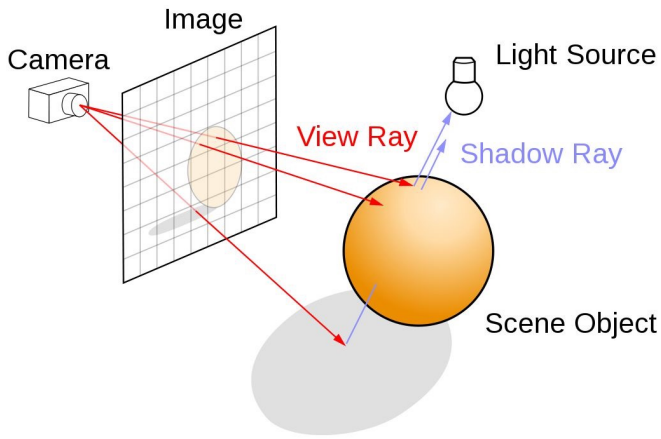

Figure 2.3: Illustration of the ray tracing process. Rays are cast from the camera, and for each pixel, the ray is traced through the scene to determine the color of the pixel. Source: NVIDIA Developer

2.2.1 Digital Differential Analysis

Due to the nature of voxel grids, the ray tracing process can be simplified by using Digital Differential Analysis (DDA). DDA is a line drawing algorithm that is used to traverse a grid, and can be adapted for ray tracing by stepping through the voxel grid along the ray using pre-calculated intervals (Amanatides & Woo 1987). Calculating the steps the ray takes through the grid before hand saves time during the ray tracing process, and can be used to determine the intersection points of the ray with the voxels. See Figure 2.4.

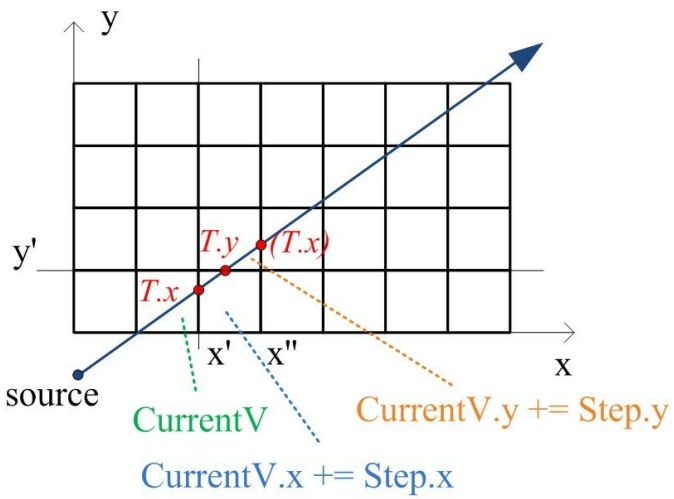

Figure 2.4: A 2D example of the DDA algorithm. Steps values for the x and y axes are calculated before hand, and used to traverse the grid. Source: (Xiao 2015)

2.2.2 Bounding Volume Hierachies

Intersection tests are an important part of the ray tracing process, and can be computationally expensive. Bounding Volume Hierarchies (BVHs) are a data structure that can be used to speed up the intersection tests by providing a way to quickly determine which objects in the scene the ray intersects with (Ize 2009). See Figure 2.5.

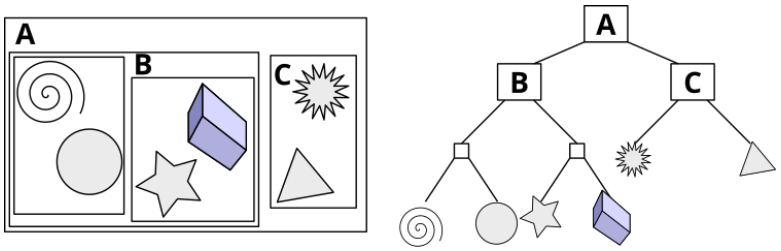

Figure 2.5: An example of a bounding volume hierarchy using rectangles as bounding volumes. Source: Wikipedia

A natural way to create a BVH for a voxel scene is to use an octree structure (see Section 2.3), where each node in the tree represents a bounding box that contains the voxels in the scene. The octree can be traversed to determine which voxels the ray intersects with, and can be used to quickly determine the intersection points of the ray with the voxels using the DDA algorithm (see Section 2.2.1).

2.3 Sparse Voxel Octrees

Sparse Voxel Octrees (SVOs) are a popular data structure for representing voxel scenes, as they provide a compact representation of the scene by storing only the occupied voxels in a tree structure. Each node in the tree can be subdivided further into 8 children to achieve a desired resolution of the scene; this property also makes it easy to introduce LOD by only ray tracing up to a specific depth in the tree. See Figure 2.6.

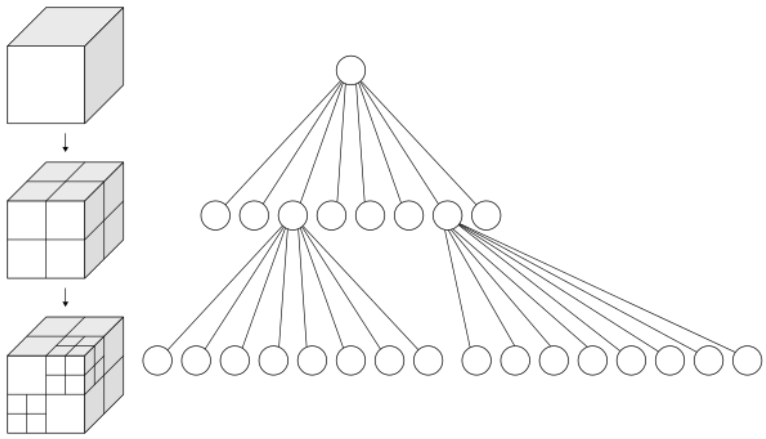

Figure 2.6: Illustration of a voxel grid represented as a sparse voxel octree. Left: Recursive subdivision of a Cubes into octants. Right: The corresponding octree. Source: Wikipedia

Octrees are most easily implemented using pointers which works well on the CPU, but can be inefficient on the GPU due to the lack of support for pointers. To address this, SVOs are usually further compressed into a format better suited for the GPU such as a 1D array, using techniques such as run-length encoding (RLE) (Eisenwave n.d.*a*) and space filling curves (SFC) (Eisenwave n.d.*b*).

2.3.1 Space Filling Curves

Space Filling Curves (SFCs) are a way of mapping a 2D or 3D space into a 1D space, and are used to linearize the octree structure of an SVO into an array. SFCs can provide better cache locality when traversing the octree, as neighbouring voxels in 3D space are likely to be close together in the 1D array. A popular SFC used in voxel rendering is the Morton code, also known as the Z-order curve. See Figures 2.7 and 2.8.

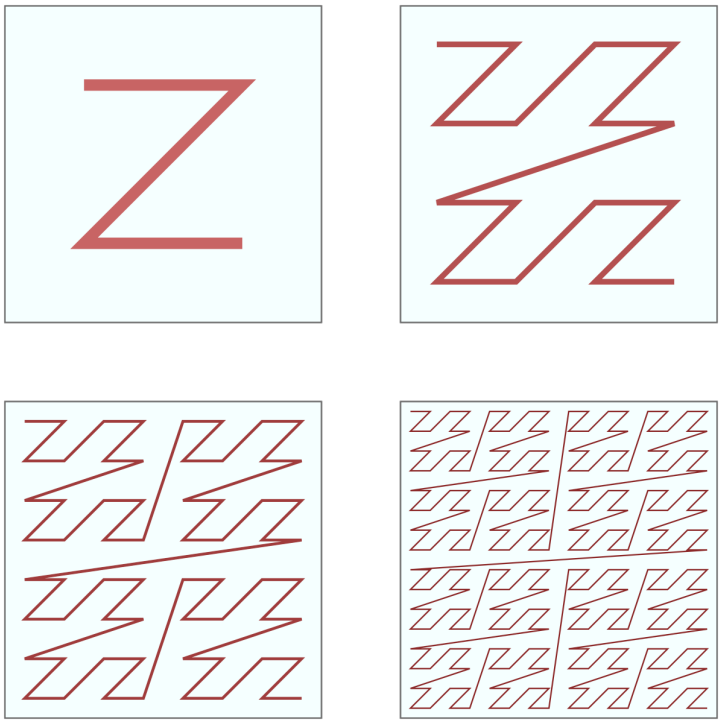

Figure 2.7: Four iterations of the Z-order curve (Morton code) on a 2D grid. Source: Wikipedia

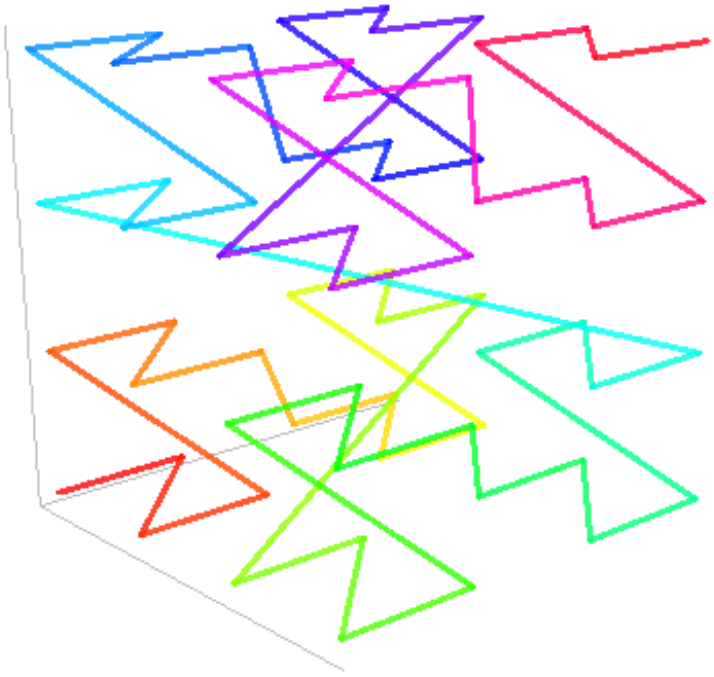

Figure 2.8: Extension of the Z-order curve (Morton code) to 3D space. Source: Wikipedia

2.3.2 Sparse Voxel Directed Acyclic Graphs

SVOs can be further optimized by using Sparse Voxel Directed Acyclic Graphs (SVDAGs) (Kampe, Sintorn & Assarsson 2013). SVDAGs are a way of further compressing an SVO by removing redundant nodes in the tree that contain identical nodes. This can lead to being able to store larger scenes in memory (in the billions of voxels (Kampe et al. 2013)) while still maintaining a low memory footprint. In the context of the dissertation, this technique could be used to store the voxel data on the GPU in a more compact format, and reduce the amount of data that needs to be transferred between the host and device; however, updating an SVDAG can be more complex than updating an SVO which would not be suited to dynamic updates.

2.4 Dynamic Updates

Updating an SVO can be a straightforward process as only the affected nodes need updating; however, updating a compressed format, and transferring the new data to the GPU, can be computationally expensive (Crassin 2012). A common approach is to divide a voxel scene into a dynamic and static SVO (Douglas 2022, Pan 2021).

There are several approaches for merging two octrees together, each of which are suited to different scenarios such as addition or removal of voxels. The two common algorithms are Jessup's and Pham's algorithms (Jessup, Givigi & Beaulieu 2014, Pham, Kim & Ko 2007).

Chapter 3

System Design

Hello.

3.1 Testing Approach

- GPU timestamps, before + after computer shader dispatch
- Distance field execution time measured in milliseconds
- Static world as the benchmark (updates disabled)
- Dynamic world using a simple falling sand simulation, sand generator top left, water generator top right
- Dynamic update test run on multiple world sizes, from 8x8x8, to 512x512x512
- Update speed limited to 10 world updates per second.
- Measure average execution time, standard deviation
- Additional measure of FPS and delta time, compare to static world to determine playability
- Vulkan application compiled in release mode + no debug or validation layers

Chapter 4

Development

4.1 Single World Buffer

Using a single large buffer for the world. An update to the world, anywhere, requires updating the whole buffer and distance field.

Table 4.1: Single World Buffer baseline frame rate and delta time performance metrics given a static world.

Buffer Size	8^3	32^3	128^3	512^3
Baseline Avg. FPS	138.63960	138.61365	132.64647	n/a
Baseline Avg. Delta Time (ms)	0.00719	0.00721	0.01853	n/a

Table 4.2: Impact of dynamic updates to the frame rate and delta time performance using a single world buffer approach.

Buffer Size	8^3	32^3	128^3	512^3
Avg. FPS	139.89711	133.22311	0.05153	n/a
Avg. Delta Time (ms)	0.00715	0.00751	19.50408	n/a

Table 4.3: Distance field compute shader execution time performance using a single world buffer approach.

Buffer Size	8^3	32^3	128^3	512^3
Avg. Time (ms)	0.12188773	10.362566	20723.77	n/a
Std. Dev (ms)	0.025453202	0.64184254	162.16953	n/a

Chapter 5

Conclusion

Hello.

Bibliography

Amanatides, J. & Woo, A. (1987), A Fast Voxel Traversal Algorithm for Ray Tracing, PhD thesis, University of Toronto.
URL: http://www.cse.yorku.ca/~amana/research/grid.pdf

Crassin, C. (2012), Dynamic sparse voxel octrees for next gen real time rendering, Technical report, NVIDIA Research.
URL: https://www.icare3d.org/research/publications/Cra12/04_crassinVoxels_bps2012.pdf

Douglas (2022), 'Sparse voxel octree modification and benchmarking'.
URL: https://www.youtube.com/watch?v=yjOLx4O634I

Eisenwave (n.d.*a*), 'Run-length encoding'. Part of the "Voxel Compression Documentation", found at the same URL.
URL: https://eisenwave.github.io/voxel-compression-docs/rle/rle.html

Eisenwave (n.d.*b*), 'Space-filling curves'. Part of the "Voxel Compression Documentation", found at the same URL.
URL: https://eisenwave.github.io/voxel-compression-docs/rle/space_filling_curves.html

Gedge, J. (2014), 'Greedy voxel meshing', *Blog* .
URL: https://gedge.ca/blog/2014-08-17-greedy-voxel-meshing/

Ize, T. (2009), Efficient Acceleration Structures for Ray Tracing Static and Dynamic Scenes, PhD thesis, The University of Utah.
URL: https://my.eng.utah.edu/~cs6958/papers/thesis_ize.pdf

Jessup, J., Givigi, S. & Beaulieu, A. (2014), Merging of octree based 3D occupancy grid maps, PhD thesis, Institute of Electrical and Electronics Engineers.

Kampe, V., Sintorn, E. & Assarsson, U. (2013), High Resolution Sparse Voxel DAGs, PhD thesis, Chalmers University of Technology.
URL: https://www.cse.chalmers.se/~uffe/HighResolutionSparseVoxelDAGs.pdf

Laine, S. & Karras, T. (2010), Efficient sparse voxel octrees - analysis, extensions, and implementation, Technical report, NVIDIA Research.
URL: https://research.nvidia.com/sites/default/files/pubs/2010-02_Efficient-Sparse-Voxel/laine2010i3d_paper.pdf

Pan, Y. (2021), Dynamic update of sparse voxel octree based on morton code, PhD thesis, Purdue University.

Pham, T., Kim, Y. & Ko, S. (2007), Development of a software for effective cutting simulation using advanced octree algorithm, *in* 'Proc. IEEE Intern. Conf. on Commun.', Kuala Lampur, Malaysia.

Xiao, K. (2015), GPU-based acceleration techniques: Algorithms, implementations, and applications, PhD thesis, University of Notre Dame.