MOXEL DAGS: CONNECTING MATERIAL INFORMATION TO HIGH RESOLUTION SPARSE VOXEL DAGS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Brent Williams

June 2015

COMMITTEE MEMBERSHIP

TITLE:	Moxel DAGs: Connecting Material Information to High Resolution Sparse Voxel DAGs
AUTHOR:	Brent Williams
DATE SUBMITTED:	June 2015
COMMITTEE CHAIR:	Zoe Wood, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	Aaron Keen, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	Shinjiro Sueda, Ph.D. Assistant Professor of Computer Science

ABSTRACT

Moxel DAGs: Connecting Material Information to High Resolution Sparse Voxel DAGs

Brent Williams

As time goes on, the demand for higher resolution and more visually rich images only increases. Unfortunately, creating these more realistic computer graphics is pushing our computational resources to their limits. In realistic rendering, one of the common ways 3D objects are represented is as volumetric elements called voxels. Traditionally, voxel data structures are known for their high memory requirements. One standard way these requirements are minimized is by storing the voxels in a sparse voxel octree (SVO). Recently, a method called High Resolution Sparse Voxel DAGs was presented that can store binary voxel data orders of magnitudes more efficiently than SVOs. This memory efficiency is achieved by converting the tree into a directed acyclic graph (DAG). The method was also shown to have competitive rendering performance to recent GPU ray tracers. Unfortunately, it does not support storing collections of rendering attributes, commonly called materials. These represent a given object's reflectance properties, and are necessary for calculating its perceived color.

We present a method for connecting material information to High Resolution Sparse Voxel DAGs for scenes, with multiple meshes, and several different materials. This is achieved using an extended Sparse Voxel DAG, called a Moxel DAG, and an external data structure for holding the material information, we call a Moxel Table. Our method is more memory efficient than traditional SVOs, and increases in efficiency in comparison when at higher resolutions. Because it stores the equivalent information as SVOs, it achieves the same visual quality at the same resolutions.

ACKNOWLEDGMENTS

Thanks to:

- Zoe Wood for supporting me through my many years at Cal Poly and always feeding my insatiable desire for graphics knowledge.
- Martin Watt, for helping me work through some of my graphics ideas longer than you ever needed to.
- Ben Chessing, for providing me the opportunity to play volleyball instead of working on my thesis, because after stepping away from it for a while is when I always had the biggest breakthroughs.
- My brother, Drew Williams, for helping me with all those late night resume and cover letter letter sessions when I knew all you really wanted to do was go to sleep.
- Finally, my parents, Gary and Julie Williams, for supporting me my whole life, especially when my dreams were on the line. I couldn't have done it without you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

In this thesis we present a method for storing and accessing material information through a graph based voxel data structure. To understand the context and motivations of our work, we must explore two questions fundamental to computer graphics: “How do we represent the world within a computer?” and “How do we turn that into an image?”

1.1 Rendering and Materials

One of the most important processes of computer graphics is creating an image from the description of a scene. This we call rendering, and most rendering algorithms are based on the physics of light.

In general, light comes from a source, reflects or refracts off of surfaces, and eventually hits our eyes. What we perceive comes from the interaction of the light with the surfaces it hits. In computer graphics, we have created models to simulate these interactions using sets of reflectance properties we call materials. Some examples of these properties include the index of refraction, which describes how much light bends when going through a surface, and the specularity, which controls the size of the bright highlight on an object based on how shiny or dull it is.

1.2 Geometry Representation

How do we represent the scene's surfaces that these material properties describe? Traditionally, they are represented as a collection of polygonal surfaces, also known as a mesh. Meshes are

made up of a discrete set of interconnected points in space allowing artists to easily manipulate them. Researchers have spent many years developing algorithms around meshes, and much of our hardware devoted to computer graphics heavily supports this paradigm. One very large problem with it is that these surfaces are inherently made of 2D shapes, and they do not very well represent the volumetric world that we live in.

In computer graphics, volumes have been relatively unfriendly to work with due to their space and algorithmic complexities. That is their inherent nature of being n^3 (where n is the size of one dimension of the volume). Because of these complexities, for many practical applications, volumes at high resolutions are unable to fit in memory, and the algorithms that work on them are relatively slow. Much work has been done to reduce these complexities, and many exciting developments have been made in recent years.

One of the most promising recent works today has been High Resolution Sparse Voxel DAGs [Kampe et al., 2013]. They present a method which stores binary voxel data orders of magnitude more efficiently than previous methods. They also present a ray tracer for their data structure that has competitive rendering performance to recent GPU ray tracers. Unfortunately, their work does not support storing materials. With the demand for more realistic renders, materials are essential for creating visually rich images.

1.3 Our Contributions

We present an extended version of a High Resolution Sparse Voxel DAG, for mid-level scenes, with multiple meshes, and several different materials, called a Material Voxel DAG (or Moxel DAG). Specifically, we add a small amount of information to DAG nodes to allow for calculating the current voxel's index into an external table of the materials that we call a Moxel Table. These

materials can be accessed while ray tracing, allowing for more realistic rendering. In this paper we will:

- Discuss the subject of voxels and associated concepts,
- Examine current voxel data structures and related works,
- Describe the extension to High Resolution Sparse Voxel DAGs, and
- Analyze our results.

While our method does require some overhead in memory and speed over Sparse Voxel DAGs, it allows for more visually rich rendering, and is much more memory efficient than traditional SVOs. This can be seen in the chapter on results, Chapter 5.

CHAPTER 2

BACKGROUND

In this work, we introduce a graph based voxel data structure, that we call a Moxel DAG, and a set of algorithms used with it for accessing material information. These are based on many fundamental concepts, and in this section we will discuss these concepts. First, we will describe ray tracing, and what acceleration data structures are. Next, we will discuss voxels and SVOs. Lastly, we will describe Morton Coding and how it is used in SVOs.

2.1 Ray Tracing

One of the most popular methods for making an image of a scene is called ray tracing. In 1968 Appel [Appel, 1968] published the ray casting algorithm where rays are shot out of a camera, through an image plane, and intersect with mathematically defined surfaces. A ray is shot through each pixel where the color of the pixel is calculated based on the reflection model of closest

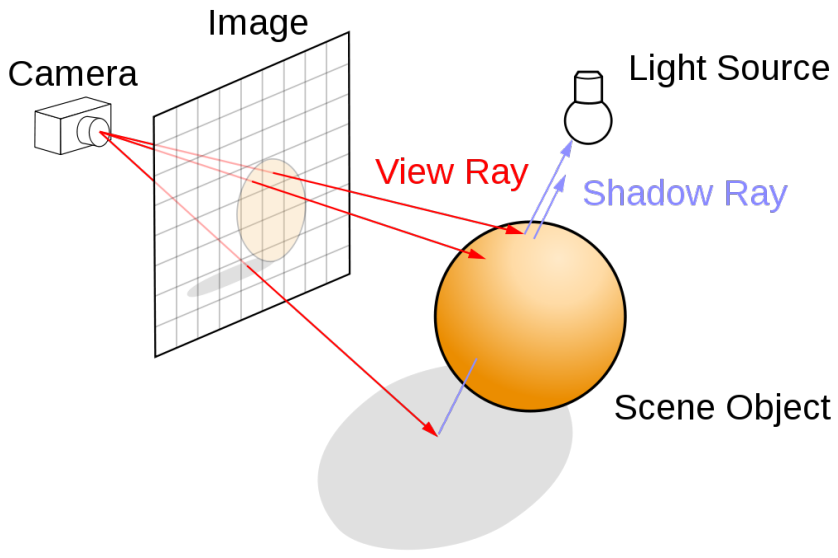


Figure 2.1: The process of ray tracing. Credit: “Henrik (Creative Commons License).

object it intersects with, and the material information of that object. A picture of the process of ray tracing can be seen in Figure 2.1.

2.1.1 The Phong Reflection Model

Reflection models are used to represent how light interacts with the surface of an object. One of the simplest and most used of these models is the Phong reflection model. It was first published in Bui Tuong Phong's Ph.D. Dissertation [Phong, 1973] in 1973. It describes light reflecting off of a surface as made up of three components: ambient, diffuse and specular. A picture of the three components can be seen in Figure 2.2 and the equation for the Phong reflection model can be seen as follows:

$$c_{final} = c_{ambient} + c_{diffuse} + c_{specular}, \tag{2.1}$$

where c_{final} is the final color, c_{ambient} is the ambient component, $c_{diffuse}$ is the diffuse component and $c_{specular}$ is the specular component.

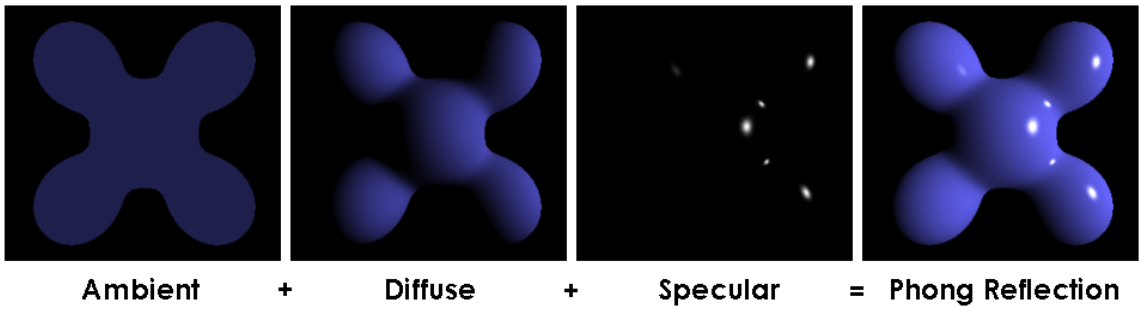


Figure 2.2: The components of the Phong reflection model. Credit: "Rainwarriorcommonswiki" (Creative Commons License).

The ambient component represents the small amount of light that bounces around the scene and comes from all directions. Its component is calculated in the following equation:

$$c_{ambient} = I_a * K_a, \tag{2.2}$$

where I_a is the intensity of the ambient light, and K_a is the ambient reflection coefficient, which is a color made up of red, green, and blue components.

Diffuse reflection is the light that bounces off of rough surfaces, and scatters equally in all directions. This is based on Lambert's Law, and can be written as the following equation:

$$c_{diffuse} = \sum_{i=0}^{j} I_i * K_d * (N \cdot L_i), \tag{2.3}$$

where $c_{diffuse}$ is the diffuse component, j is the number of lights, I_i is the intensity of light i, and K_d is the diffuse reflection coefficient, which is a color made up of red, green, and blue components. N dot L_i represents how small the angle is between the vector N and the vector L_i where N is the normal, or the perpendicular vector to the surface at that point, and L_i is the direction of the light i's vector.

Specular reflection is the light that bounces off of smooth surfaces and is the most intense in the direction of the reflection coming from the light and bouncing off of the surface. This can be calculated as follows:

$$c_{specular} = \sum_{i=0}^{j} I_i * K_s * (V \cdot R_i)^{\alpha}, \tag{2.4}$$

where $c_{specular}$ is the specular component, j is the number of lights, I_i is the intensity of light i, and K_s is the specular reflection coefficient, which is a color made up of red, green, and blue components. α is the specularity, and controls the size of the specular highlight. V dot R_i represents how small the angle is between the vector V and the vector R_i where V is the vector pointing from the point on the surface to the viewer, and R_i is light i's reflection vector. R_i can be calculated as follows:

$$R_i = 2 * (L_i \cdot N) * N - L_i, \tag{2.5}$$

where *Ri* is the reflection vector for light *i*, *Li* is the direction of the light *i*'s vector, and *N* is the surface normal at that point.

A condensed equation for the whole Phong reflection model can be seen as follows:

$$c_{final} = (I_a * K_a) + \sum_{i=0}^{j} (I_i * K_d * (N \cdot L_i)) + (I_i * K_s * (V \cdot R_i)^{\alpha}). \tag{2.6}$$

2.1.2 From Ray Casting to Ray Tracing

It was not until 1980 that Whitted [Whitted, 2005] extended the idea of ray casting to continue processing for a pixel at the site of intersection. He suggested a ray can be cast from the first intersection point to the light to test if it is in shadow. He also suggested that rays could reflect or refract recursively throughout the scene based on the material properties of the object that it hits such as index of refraction, and reflection coefficient.

2.1.3 Ray Tracing Advantages and Disadvantages

Over other rendering algorithms, ray tracing has many advantages but it also suffers from many performance issues. Using a brute force method, for each ray, intersection testing needs to be done with every object in the scene, leading to a performance of O(n). To increase the performance, acceleration data structures can be used. These spatially subdivide the scene allowing faster intersection testing. One such acceleration data structure is a sparse voxel octree (SVO).

2.2 Sparse Voxel Octrees

Just about all computer images today are made up of pixels. The word *pixel* is a combination of the two words “picture” and “element.” Like pixel, the word *voxel* is a combination of two words: “volume” and “pixel.” So, a voxel is a 3D volume element that usually has the same length, width and height. They usually do not store their 3D coordinates explicitly, but they are inferred based on their position relative to others.

The process of converting a 3D scene represented by another geometry format into one represented by voxels is called voxelization. This is usually accomplished by testing whether the scene geometry intersects with voxels, and if so, storing some of that geometry's information within those voxels.

Voxels can be stored as a regular grid, but especially for sparsely filled scenes, it wastes a lot of space on empty voxels. To combat this problem, sparse voxel octrees (SVOs) were proposed. If the entire scene was contained within a voxel, and it was split down the voxel's center along the X, Y, and Z axis, the result would be 8 smaller child voxels. Until the desired resolution is reached, or a voxel is completely empty, each voxel can be split into 8 child voxels. This makes an octree, and is the basic idea of an SVO. Because the SVO stops subdivision at the largest empty voxel node, and most scenes are relatively sparse, SVOs can potentially save a lot of space. Also, because at each level, the octree spatially subdivides the scene, it acts as an acceleration data structure, which increases ray tracing performance. A picture of an SVO and its 3D representation can be seen in Figure 2.3.

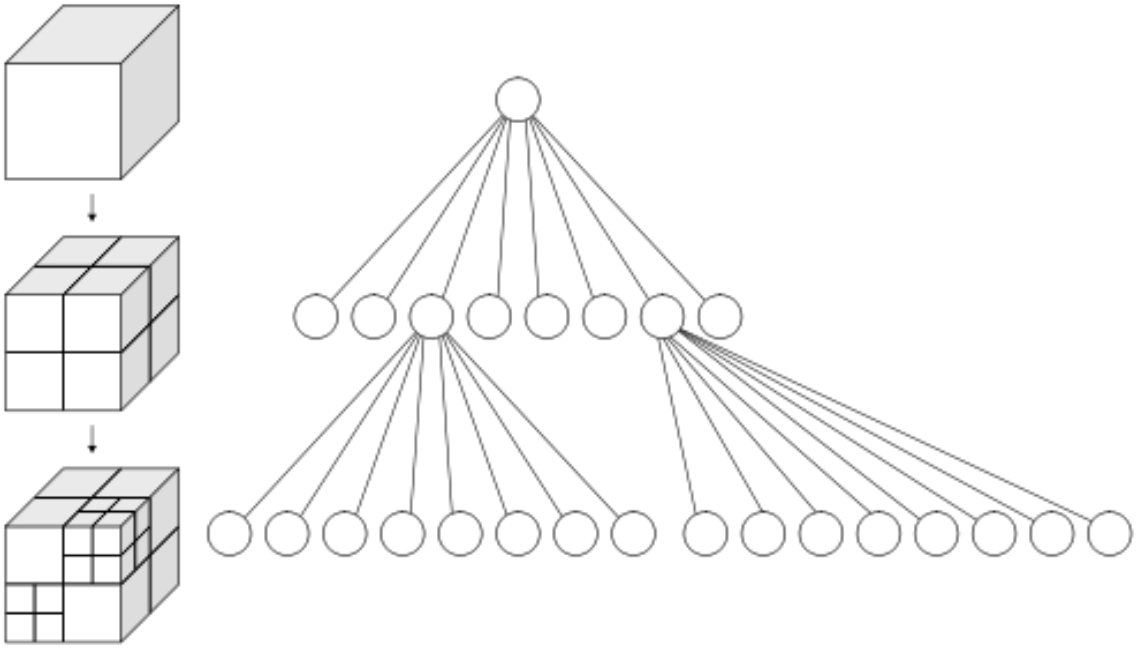


Figure 2.3: A SVO and its 3D representation. Credit: “WhiteTimberwolf” (Creative Commons License).

The smallest voxels at the bottom of the tree are called leaf voxels. Material information or a reference to the material information is usually stored in the leaf voxels. It can also be stored in the voxel nodes higher than the leafs, where these represent the combined material information of all its children. Doing so allows for optimizations using level of detail.

2.3 Morton Coding and Z-Order

When building SVOs, it is common to have an established ordering of all child nodes. One of the most popular is to use the ordering defined by Morton coding. Morton coding maps multidimensional space (in our case 3D), to one dimension while preserving spatial locality. If applied to voxels, the order of the voxels in 3D space will make z shaped patterns. This is why Morton order is also known as Z-order. A 2D example can be seen in Figure 2.4, and a 3D example can be seen in Figure 2.5.

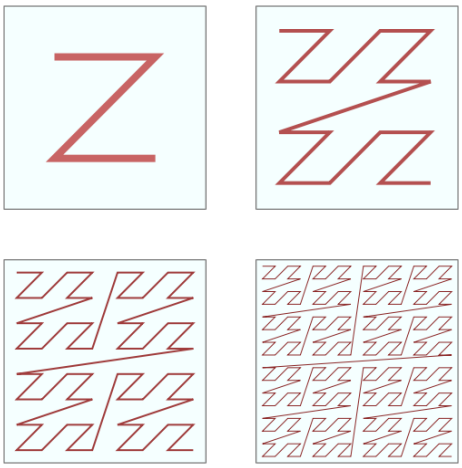


Figure 2.4: 2D Morton Code at 4 different levels. Credit: "David Epstein" (CreativeCommons License).

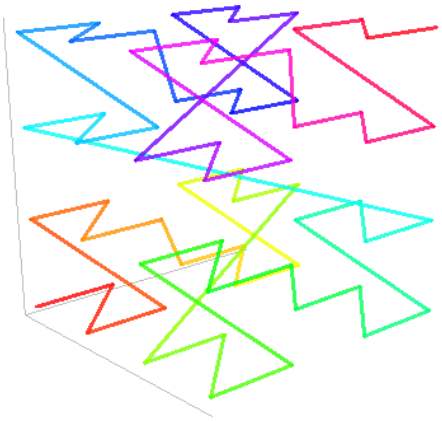


Figure 2.5: 3D Morton Code. Credit: "Robert Dickau" (Creative Commons License).

To convert a set of (x, y, z) integer coordinates to a Morton code, the x, y and z values should be converted to binary and then their bits should be interleaved. For 3D coordinates, if only eight values need to be ordered, only three bits are needed. For every power of eight, another three bits are needed. An example of interleaving bits for Morton encoding can be seen in Figure 2.6. Because of its spatial locality, Morton coding improves caching, and is used as the ordering for child voxels in most SVOs.

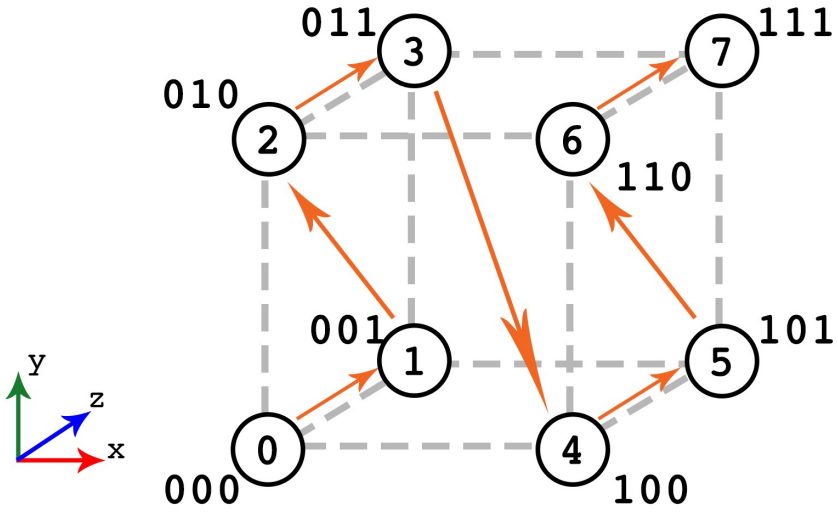


Figure 2.6: Bit interleaving for 3D Morton encoding.

CHAPTER 3

RELATED WORK

In this paper, we present a method for connecting material information to High Resolution Sparse Voxel DAGs for mid-level scenes, with multiple meshes, and several different materials. Research into storing and rendering voxels has been going on for many years. In this chapter we will discuss some of this work focusing on the uses of voxel trees, the history of subtree merging, rendering octrees, and tree based voxel data structures used for rendering.

3.1 Voxel Trees and Their Uses

Voxel tree data structures have been used for many years and for several different purposes. In 2005, Gobbetti and Marton [Gobbetti and Marton, 2005] suggested replacing objects with voxels when viewed from far away and stored them in a tree. In “Octree-Based Progressive Geometry Coding of Point Clouds”, Huang et. al [Huang et. al, 2006] provide an algorithm for encoding and compressing point cloud data by discretizing it into an octree.

3.2 Merging Common Subtrees

Our method heavily relies on the idea of merging common subtrees. This is not a new idea, and has been applied before. In 1989, Webber and Dillencourt [Webber and Dillencourt] published a paper on compressing binary images by representing them as quadtrees and merging common subtrees. This idea was later extended to 3D in a paper presented by Parker et. al [Parker and Udeshi, 2003] in 2003 in order to compress voxel data. Their proposed method does not decouple

surface properties from the nodes, and the authors only report results on electrical circuits which are flat and very regular. Many other methods of compressing trees can be seen in this survey paper [Katajainen and Makinen, 1990] by Katajainen et. al.

3.3 Rendering Voxels and Octree Traversals

Since the advent of voxels, there has been a desire to render them. The first to present a regular grid traversal algorithm was Amanatides and Woo [Amanatides and Woo, 1987] in 1987. Their work is what most voxel traversal algorithms are derived from today. When rendering an octree full of voxels, one of the most frequent operations is ray-octree intersection testing, and in 2000, Revelles et. al [Revelles et. al, 2000] publish a paper on how to efficiently do that.

Later in 2006, Knoll et. al [Knoll et. al, 2006] presented a method for ray tracing volumetric data stored in a octree that is visualized at several isosurface levels. In 2009, they extend their method to include coherent packet ray tracing [Knoll et. al, 2006]. There has been much more research on rendering voxels and octrees. The most related to our work is described in the next section.

3.4 Tree-Based Voxel Data Structures Used for Rendering

Over the course of computer graphics history, there have been many proposed tree-based voxel data structures used for rendering. Some of the most recent today include GigaVoxels, ESVO, VDB, and High Resolution Sparse Voxel DAGs.

3.4.1 GigaVoxels

Crassin et. al [Crassin et. al, 2009] presented GigaVoxels at I3D in 2009. It is a fast GPU based voxel ray tracing solution. Their method utilizes a tree based structure that is adapted at render time to have the necessary resolution to render in real time.

A constant size of tree nodes, usually 32^3, are stored together. Each node stores a pointer to its children, and a pointer to a 32^3 set of leaf voxel colors, known as a brick, or a single color if all its leaves have the same color. A full tree is saved in main memory, or if too large, on disk. Portions of the tree are loaded onto the GPU as needed, swapping out for the least recently used nodes.

One of the problems with this solution is that these bricks waste a lot of space representing empty nodes. That being said, 3D hardware texture lookups of these bricks allow for very efficient access and automatic anti aliasing. Another problem is that their solution necessitates frequent and very expensive data transfers between the CPU and GPU that other methods do not need.

3.4.2 ESVO

Efficient Sparse Voxel Octrees (ESVO) was fully released as an in depth, technical paper in 2010 by Laine and Karras [Laine and Karras, 2010]. They propose a sparse voxel octree that can be efficiently ray-casted on the GPU. Contour data is saved within voxels to help better approximate the edges of surfaces and reduce the blockiness that plagues most voxel surface representations. A compression scheme for material information that stores per voxel an average of 1 byte for colors

and 2 bytes for normals was also presented. This allows for huge memory savings for material information.

Due to memory requirements, they could not keep the whole tree on the GPU and thus had to keep a copy of the tree in main memory and stream portions of the tree to the GPU on demand. While they did save contour data, without post processing, visual artifacts were still evident.

3.4.3 VDB

VDB is a high-resolution sparse volume data structure and set of algorithms published by Ken Museth [Museth, 2013] of DreamWorks Animation in 2013. It has been adopted by much of the visual effects industry because of its flexibility and speed.

It is similar to a B+ tree in that it is a n-ary tree with a variable but usually large number of children per node. The root node is sparse and resizable, where all other nodes are dense and have branching factors that decrease and are restricted to powers of 2. This structure, along with its complex algorithms, allow for ease of dynamic changes, relatively good memory efficiency, fast random access, fast sequential access, configurability, and adaptive resolutions.

While this proposal does seem very promising, it is not without drawbacks. Its data structure and algorithms are somewhat complex to implement. Also, it is not the most memory efficient solution but this is made up for in its very fast access times and flexibility.

3.4.4 High Resolution Sparse Voxel DAGs

High Resolution Sparse Voxel DAGs was presented at SIGGRAPH 2013 by Kampe et. al [Kampe et. al, 2013]. It is a method and data structure for storing binary voxel data orders of magnitude more efficiently than SVOs.

An SVO can be implemented such that the scene is represented by an N^3 binary voxel grid. The octree is recursively subdivided L times where $N = 2^L$ and L is known as the maximum level. Each subtree of the last two levels has a resolution 4^3 and is stored in a 64 bit integer, where a leaf voxel is full if its bit is '1' and empty if it is '0.' We will call these leaf nodes. Nodes above that level are represented by an 8 bit child mask, where each bit indicates whether that child is full or empty, and a list of the pointers to non-empty children. An example of this can be seen in Figure 3.1.

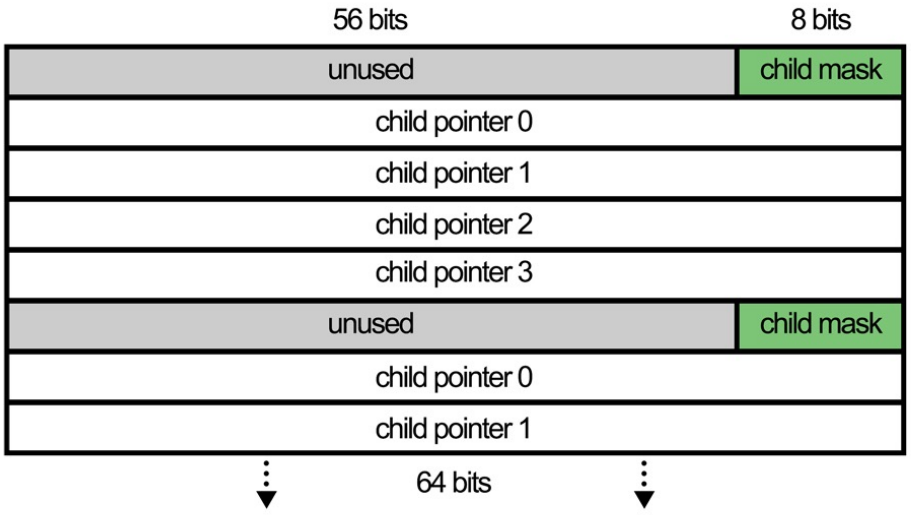


Figure 3.1: DAG nodes in memory. For each node, 8 bytes are used for the child mask, and 8 bytes are used for each pointer to a non-empty child. The child mask is given 64 bits so that it is byte aligned with the child pointers, but it only uses 8 bits, leaving 56 bits unused.

Because no spatial information is stored in the SVO, it is the unique path taken that identifies each voxel. Consider a node in the described SVO. If two of its children have all the same child masks in their subtrees and the same leaf nodes, those two children would provide equivalent paths down the tree. Because those children provide equivalent paths, they could point to the same node. Since, multiple children can now point to the same node, the octree becomes a directed acyclic graph (DAG).

To convert an SVO to a DAG the authors suggest starting at the bottom of the tree, merging identical leaf nodes, and updating their parent's child pointers. To process the level above that, we merge nodes with the same child masks and child pointers. Their parent nodes can then be updated to point to the merged nodes. This process can be repeated for all the remaining levels up to the root. An example of this process can be seen in Figure 3.2.

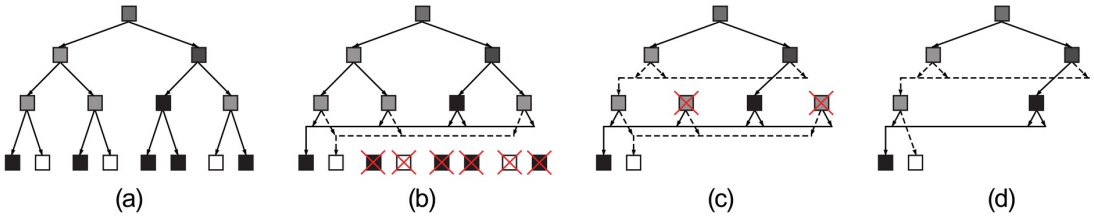


Figure 3.2: The process of reducing a SVO shown using a binary tree. a) The algorithm starts with the SVO. b) Duplicate leaf nodes are merged and leaf node parents are updated.c) The duplicates in the level above the leafs are merged and parent nodes are updated. This process is repeated until the root node is reached. d) The resulting Sparse Voxel DAG.

In their testing, the number of decreased nodes resulting from the reduction process ranged from 28X for a highly irregular scene, all the way up to 576X for a more regular scene. This allowed them to store very high resolution scenes on commodity graphics cards. When ray traced on the GPU the Sparse Voxel DAG was faster than an ESVO and comparable at high

resolutions to an industry standard triangle ray tracer. While these are all great results, Sparse Voxel DAGs are not without their drawbacks. Arguably, the largest drawback is the lack of support for materials, which is what our work with Moxel DAGs is trying to solve.

CHAPTER 4

HIGH RESOLUTION SPARSE VOXEL DAG EXTENSION

In this chapter, we present our extended version of a High Resolution Sparse Voxel DAG for rendering 3D voxelized scenes. Specifically we add material information which is essential for rendering visually rich images. We call our extension a Moxel DAG. It can be used for scenes with multiple meshes, and several different materials. We accomplish this by adding a small amount of information to DAG nodes to allow for calculating the current voxel's index into an external table of the materials, that we call a Moxel Table. This extension stores equivalent information to a traditional SVO and, together with the presented algorithms, allows for equivalent renders at the same resolutions while requiring much less memory.

4.1 The High Resolution Sparse Voxel DAG Algorithm

In general, High Resolution Sparse Voxel DAGs represent the scene as a DAG that saves memory by only keeping one copy of equivalent children at the same level. A simplified version of the DAG building process is as follows:

1. A scene made up of triangle meshes is voxelized.
2. A SVO is built from the voxels with every nodes children being sorted in Morton order, and the last two levels being represented by unsigned 64 bit integers.
3. The SVO is then reduced to a Sparse Voxel DAG.
    (a) The leaves are sorted and reduced to unique leaves.

(b) The parents of the leaf nodes are then updated to point to the reduced unique leaves.

(c) The nodes of the level above that are sorted based on their child pointers and reduced to unique nodes.

(d) The parents of those nodes are then updated to point to their reduced unique children.

(e) This process, starting with step (c), is repeated until the whole tree is reduced.

The resulting DAG then can be traversed and ray traced using similar if not the exact same methods as SVOs.

4.2 Extension Overview

We use the same process for building the DAG as described in the previous section, except we add additional steps at the end to convert it into a Moxel DAG and store the material information. Specifically we augment the DAG nodes to keep track of the number of empty voxels that would be between filled nodes. The resulting Moxel DAG allows the algorithm, during traversal, to calculate an offset into a table that only includes material information for filled nodes. We call this table the Moxel Table.

4.3 Shading Model and Material Information

The reflection model chosen for the scene specifies the material properties that need to be stored. In this thesis, we use the Phong reflection model. Other reflections models could be used as desired with small changes to the material data stored in the Moxel DAG. For this model, the material properties include the normal, the ambient reflection coefficient, the diffuse reflection coefficient, the specular reflection coefficient, and the specularity. Note that more values are needed for calculating the reflected color at a given point, but these are either stored separately as properties of the scene (i.e. the intensity values of the lights), or we have chosen to recalculate these values (i.e. the view, and light vectors) when they are needed.

4.4 Accessing the Material Information

In a typical SVO, the material information, or a reference to it, is kept in the nodes of the tree so that when the final destination node is reached, it can be directly accessed. This is possible because there is a one-to-one correspondence between voxels in the scene and nodes in the tree. Unfortunately this is not true for Sparse Voxel DAGs. Because multiple parents are allowed to point to the same child, and the same parent is allowed to have multiple children that reference the the same node, one-to-one correspondence is lost. This means each node in the DAG might represent one or more voxels in the scene. Graphically this means that multiple portions of the scene share the same geometry, but are in different locations. Unfortunately, they are not guaranteed to share the same material information.

Let us assume that we will keep an external table of material information where each voxel in the scene has its own place in the table. To calculate the index of the material information for the current voxel, we need a way to map a voxel in the leaf node of the DAG to an index. This can easily be done because we utilized Morton Coding when building the DAG.

4.4.1 Calculating the Voxel's Index

A voxel's index is the Morton index assigned to it, which can be calculated from an SVO. An SVO node's child mask and child pointers are indexed in morton order. That means that if a two level SVO had all of its voxels full, and the tree was visualized, the index zero leaf would be on the far left and the index 63 leaf would be on the far right, as seen in Figure 4.1.

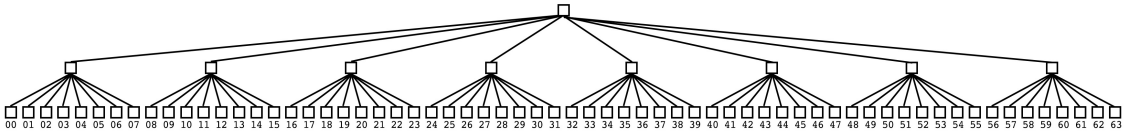


Figure 4.1: The indexes of leaf voxels of a two level SVO, with all voxels full, given in morton order.

To determine the voxel index, a running sum is kept during traversal. Every time a child is traversed to, the value added to the running sum is calculated using the following equation:

$$v = c * \left(8^{n-l-1}\right), \tag{4.1}$$

where *v* is the value added to the running sum, *c* is the child index that is being traversing to, *n* is the number of levels in the SVO not counting the root, and *l* is the index of the current level with zero being the root and *n - 1* being the leaves. This value represents the number of voxels in the left subtrees. An example of calculating the index for a voxel upon traversal can be seen in Figure

4.2. It is the exact same calculation for a Sparse Voxel DAG. This is because the traversal on a DAG is equivalent to the traversal on an SVO.

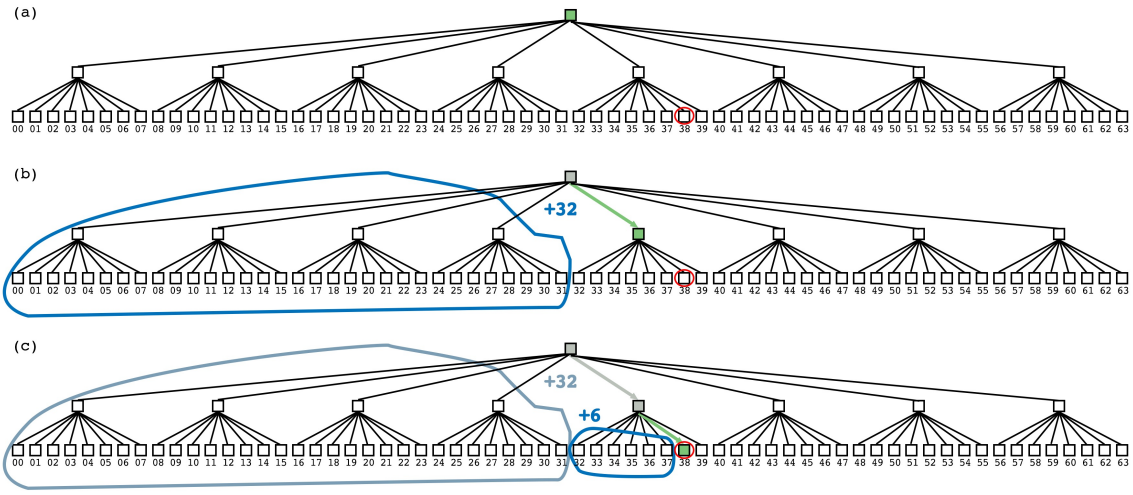


Figure 4.2: An example of calculating the voxel index on traversal. The green node is the current node, and the node circled in red is the node being traversed to. (a) The algorithm starts at the root, level 0. (b) It traverses down to the index 4 child, adding to the index sum $4*(8^{2-0-1}) = 32$. (c) It then traverses down to the index 6 child adding $6*(8^{2-1-1}) = 6$ to the index sum, giving a final voxel index of 38.

Now that the voxel index is calculated, the voxel's material information can finally be accessed. While this method works, it requires the computer to save space for material information for every voxel in the scene whether it is empty or full. The vast amount of space needed for this method makes it impractical. Instead, we can use a table that only needs to keep material information for filled voxels, which we will call a Moxel Table. This saves memory, but how is the index into the Moxel Table calculated? Two numbers are necessary: the voxel index, and the number of empty nodes before that voxel.

4.4.2 Calculating the Number of Empty Voxels

When calculating the number of empty voxels, it is necessary to keep a running sum while traversing to child nodes. The value added to the running sum is the number of empty voxels in the left subtrees. One possible method for calculating this value is to traverse all the left subtrees whenever a child node is visited. Then the number of empty voxels could be counted using a similar technique to calculating the voxel index. While this works, it is very inefficient. What can be done instead is pre-calculating the empty voxel counts for each subtree and storing them in the parent of their node. We replace Sparse Voxel DAG nodes with these modified nodes. The resulting structure we call a Moxel DAG. With empty voxel counts stored in each node, the value added to the running sum can be calculated by the following equation:

$$v = \sum_{i=0}^{k-1} e_i, \tag{4.2}$$

where v is the value added to the running sum, k is the index of the child that is being traversed to, and e_i is the empty count for the ith child.

4.4.3 Calculating the Moxel Index

Given now that we have the voxel index and the number of empty nodes before that voxel, we can now calculate the index into the Moxel Table. We will call this the moxel index, and it can be calculated as follows:

$$m = v - e, \tag{4.3}$$

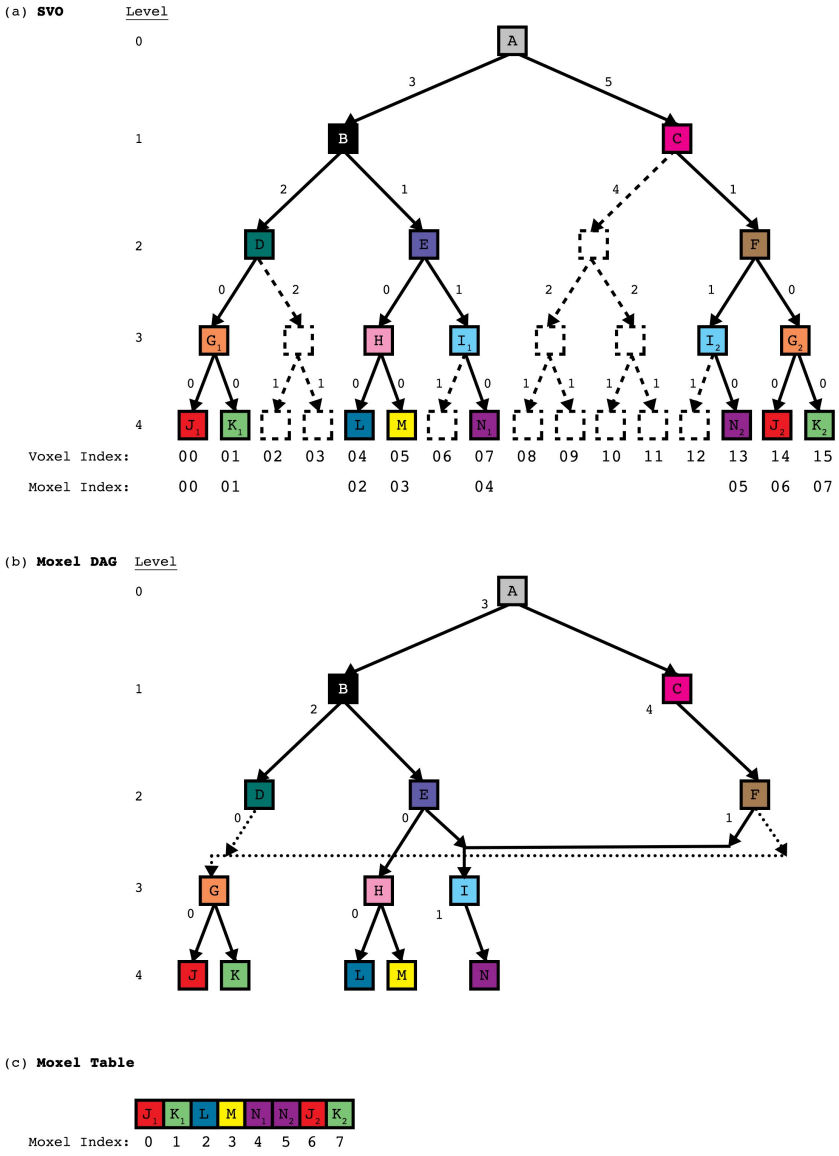


Figure 4.3: (a) An SVO illustrated as a binary tree for simplicity. The numbers on the edges represent the empty counts for that subtree. These are not actually stored in the SVO but shown for ease of comparison with the Moxel DAG. (b) The Moxel DAG reduced from the SVO in a, with the empty counts for each node's left subtree. Note a Moxel DAG reduced from a binary tree will only need one empty count per parent node, where a octree will need up to 7 for the 7 possible left subtrees. (c) The Moxel Table for the Moxel DAG in b given the SVO in (a).

where m is the moxel index, v is the voxel index, and e is the number of empty nodes before that voxel. An example of a SVO, its corresponding Moxel DAG and Moxel Table can be seen in Figure 4.3. An example of calculating a moxel index can be seen in Figure 4.4.

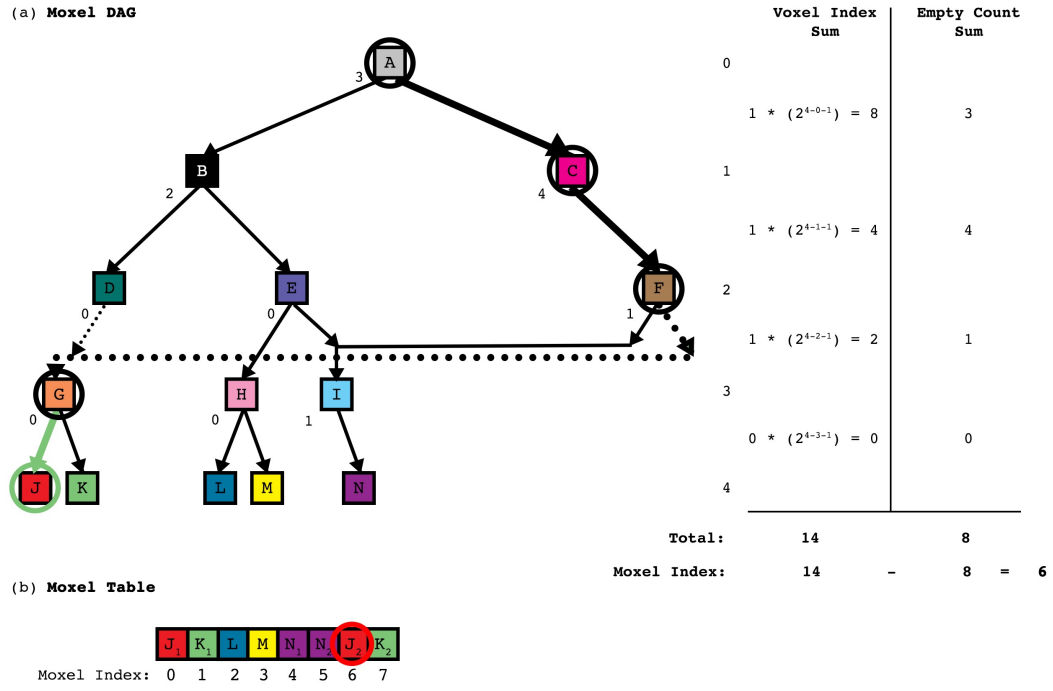


Figure 4.4: An example of calculating the moxel index for J_2. a) The same Moxel DAG and b) the same Moxel Table as Figure 4.3.

4.5 Implementation Details

In the following sections we will go over the implementation details including the Moxel DAG node structure, and building the Moxel DAG and Moxel Table.

4.5.1 Moxel DAG Node Structure

A Sparse Voxel DAG node has two main parts: a mask, and a set of pointers to its non-empty children, as seen in Figure 3.1. To keep byte alignment, a mask is allocated eight bytes while only needing one. This leaves seven bytes unused. A Moxel DAG node is structured similarly to a Sparse Voxel DAG node, except Moxel DAG nodes take advantage of the seven unused bytes for storing empty voxel counts. Empty voxel counts need to be stored for every filled child except the rightmost filled child, because they are not used in the empty voxel calculation. An example Moxel DAG node can be seen in Figure 4.5.

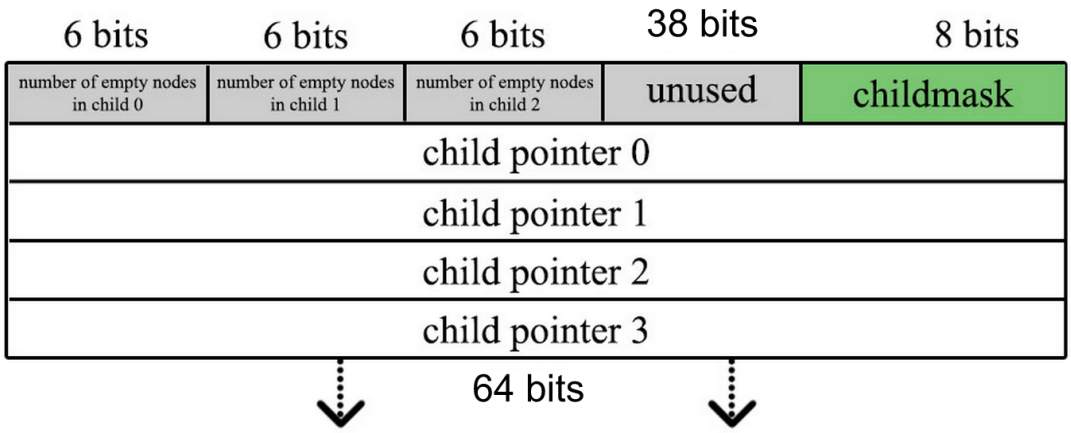


Figure 4.5: An example of a Moxel DAG node.

For Moxel DAGs above three levels, extra 64 bit chunks are allocated to fit the empty voxel counts. The number of chunks necessary varies per level and can be seen in Table 4.1. Note that because a Moxel DAG is above three levels does not mean that every levels' nodes need extra 64 bit chunks allocated. The space for empty counts can be resized per level to fit the minimum number of 64 bit chunks necessary to fit the maximum empty counts. Also note that the higher in

the Moxel DAG the node is, the more 64 bit chunks will be necessary to fit the empty counts, but also note that the higher in the Moxel DAG the node is, the less nodes there can be.

Levels	Bits Per Non-First Child	Total Needed Bits	64 Bit Chunks Added
3	6	42	0
4	9	63	1
5	12	84	1
6	15	105	1
7	18	126	2
8	21	147	2
9	24	168	2
10	27	189	3
11	30	201	3
12	33	231	3

Table 4.1: A table listing the number of bits needed for empty counts at specific levels above the leafs.

4.5.2 Building the Moxel DAG

The method implemented to build a Moxel DAG is the exact same as described in Section 4.1 except further processing is done when the DAG nodes are created. After all child nodes are reduced in the entire SVO, one more pass is done going from top to bottom to convert the SVO nodes into Moxel DAG nodes. For each node, this process includes allocating space for a mask,

the empty counts, and the child pointers. Once that is done the SVO is now completely converted to a Moxel DAG.

4.5.3 Building the Moxel Table

Originally when voxelization is done, a table is created mapping filled voxels to their corresponding triangles. Next, each filled voxel is iterated through in Morton order. For each of those voxels, its corresponding triangle in the map is accessed and material information is stored. As stated before, this material information includes the normal, the ambient reflection coefficient, the diffuse reflection coefficient, the specular reflection coefficient, and the shininess factor.

To save more space, we realized that much of the material information is the same for voxels representing the same mesh. One copy of each of these sets of material properties can be stored in a separate table which we call the Material Table. For our scenes, these properties include the ambient reflection coefficient, the diffuse reflection coefficient, the specular reflection coefficient, and the specularity. Thus for each filled voxel, the Moxel Table only stores the normal and an index into the Material Table.

CHAPTER 5

RESULTS

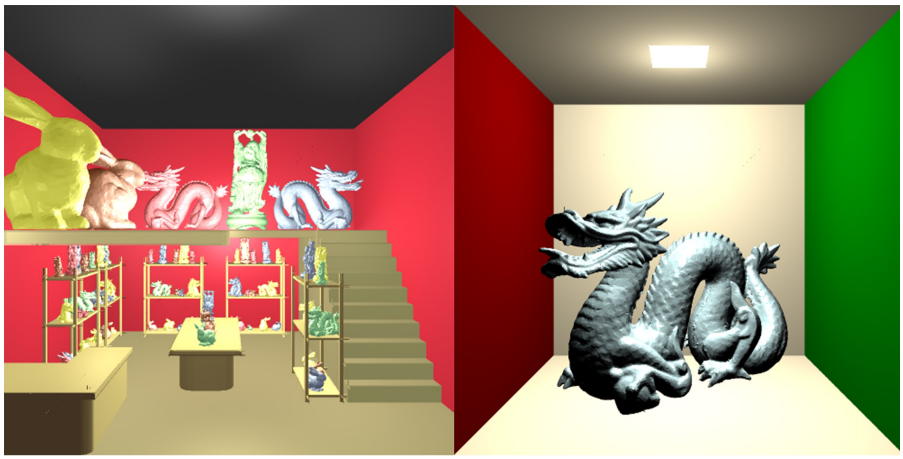
We have introduced new data structures to store, and access material information used when rendering a complex scene with multiple meshes with various materials. We also have introduced the algorithms necessary for building these data structures. In this section, we will discuss the test scenario used to compare Sparse Voxel DAGs, and traditional SVOs to our Moxel DAG data structure, analyze the results from those tests, and conclude our work.

5.1 Test Environment

All tests were done on an Apple MacBook Pro running OS X 10.10.4, with 16GB of RAM, and a 2.5 GHz Intel Core i7 processor that has four physical cores. All code was written in C++, with any multithreading done using Threading Building Blocks.

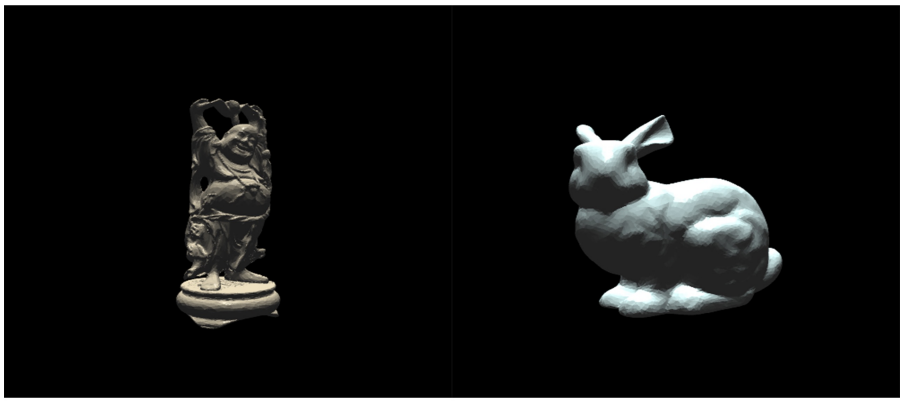
5.2 Test Scenes

We use four scenes to test our work: Toy Store, Cornell Box, Buddha, and Bunny. Toy Store is a 714,211 triangle scene with eight different materials. It is made up of relatively simple geometry except for the toys themselves. These include the Stanford bunny, Buddha, and dragon. Each has been reduced so that the Stanford bunny has 4,968 triangles, the Buddha has 12,516 triangles, and dragon has 12,500 triangles. Cornell Box is the typical box with red, green and tan walls, and has the Stanford dragon in it. The scene is 100,012 triangles with 100,000 of them being a remeshed version of the dragon itself and it has five materials. Buddha is a remeshed Stanford Buddha made up of 100,000 triangles and one material. Bunny is a remeshed Stanford bunny made up of



a) Toy Store
(714,211 triangles,
8 materials,
8 point lights)

b) Cornell Box
(100,012 triangles,
5 materials,
5 point lights)



c) Buddha
(100,000 triangles,
1 material,
5 point lights)

d) Bunny
(29,808 triangles
1 material,
5 point lights)

Figure 5.1: The scenes used in our testsT. All the images are ray traced using Moxel DAGs.

29,808 triangles and one material. Toy store is rendered with eight point lights, and all other scenes are rendered with five. The scenes can be seen in Figure 5.1, and a table of their triangle counts along with other scene data can be seen in Table 5.1.

Scene	Number of Triangles	Number of Materials	Number of Point Lights
Toy Store	714,211	8	8
Cornell Box	100,012	5	5
Buddha	100,000	1	5
Bunny	29,808	1	5

Table 5.1: A table listing the number of triangles, the number of materials, and the numberof point lights that are in each scene.

5.3 Data Structures

In our testing we compare three different data structures: Moxel DAGs, Sparse Voxel DAGs, and traditional SVOs. We also include the sizes of the generated Moxel Tables. The specifics of each are described below.

The Moxel DAGs are made as explained in Chapter 4 where all levels but the leaf level are made up of Moxel DAG nodes. These nodes will have at least one *uint64_t* (i.e. an unsigned 64 bit integer) allocated for the mask and the first bits of the empty counts. Zero or more *uint64_t*s are allocated to fit the minimum number of bits needed for the empty counts at that level. Pointers that are also 64 bits will be allocated for each non empty child. The last two levels of the DAG are represented by *uint64_t*s, where each bit represents whether a voxel is empty or full.

The Sparse Voxel DAGs are structured almost identically to as described in the paper [Kampe et. al, 2013], with the exception of having 64 bit pointers because we are running the code on a 64 bit machine. All levels but the leaves are made up of DAG nodes. These nodes

contain a child mask allocated 64 bits, and 64 bit pointers for every non empty child. The last two levels are represented by *uint64_t*s.

As for the SVOs, all levels but the leaves are made up of eight 64 bit pointers to their children. Similar to the Sparse Voxel DAGs and Moxel DAGs, the leaves represent the last two levels, and are made up of *uint64_t*s.

The Moxel Tables are made up of a normal (three *float*s of 32 bits each), and an index into the table of materials (one *unsigned int* of 32 bits) for every filled voxel in the scene.

Note that the voxel resolutions chosen for testing our scenes were limited to an upper bound of 1024^3 due to memory constraints caused by the method implemented for voxelization, and building the Moxel Tables.

		Memory in KB		
Scene		256^3	512^3	1024^3
ToyStore	Moxel DAG	254	837	2,896
	Sparse Voxel DAG	210	672	2,269
	SVO	910	4,077	17,042
CornellBox	Moxel DAG	196	726	2,609
	Sparse Voxel DAG	161	574	2,000
	SVO	256	3,168	12,980
Buddha	Moxel DAG	104	487	1,805
	Sparse Voxel DAG	126	389	1,403
	SVO	789	3,304	13,568
Bunny	Moxel DAG	177	639	2,261
	Sparse Voxel DAG	144	498	1,713
	SVO	851	3,555	14,513

Table 5.2: Comparison of memory requirements for Moxel DAGs, Sparse Voxel DAGs andSVOs. Voxel resolutions are listed in the second row and result from 8, 9, and 10 level graphs respectively.

	Memory in MB		
Scene	256^3	512^3	1024^3
ToyStore	10	38	157
CornellBox	7	38	113
Buddha	7	30	118
Bunny	8	31	125

Table 5.3: Moxel Table sizes for different resolutions of the different scenes.

5.4 Analysis

Our analysis involves comparing the memory requirements, building times, and rendering times for the aforementioned data structures. We also will discuss any possible limitations of our work.

5.4.1 Memory

Our results for memory requirements can be seen in Table 5.2 and Table 5.3. In all tests Moxel DAGs provided much smaller memory requirements than the traditional SVO. As seen in Figure 5.2, as the resolution gets larger, the Moxel DAG in comparison to the SVO takes up a smaller percentage of space. This is due to the opportunities for reducing subtree copies in the DAG as the resolution gets larger.

As seen in Figure 5.3, as the resolution gets larger, the Moxel DAG in comparison to the Sparse Voxel DAG takes up a larger percentage of memory. This is due to the increasing memory requirements for holding the empty counts at each level as the resolution increases.

While the presented method does allow for material information, it unfortunately does not compress any of that data. As seen in Table 5.3 and Figure 5.4, the memory needed to represent the material information increases by about a factor of four for every level of the tree

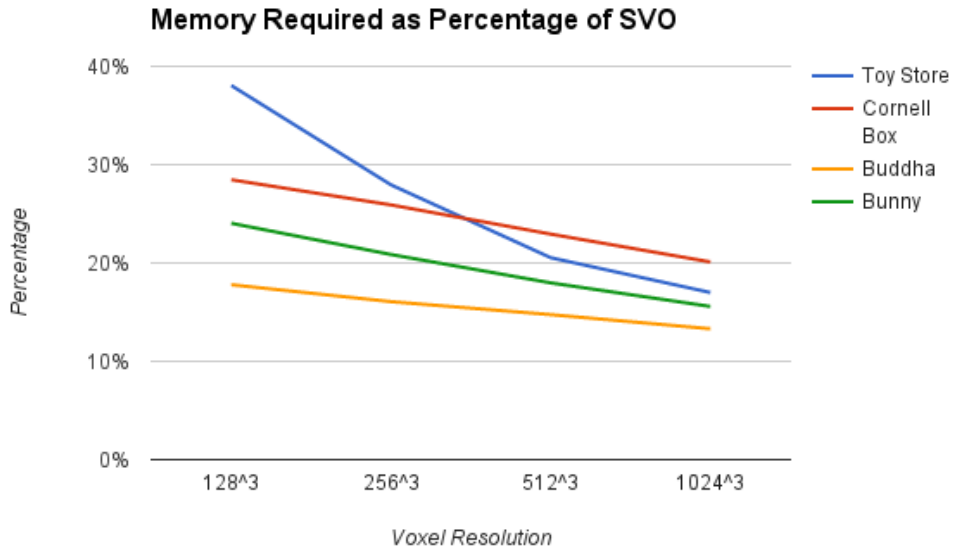


Figure 5.2: A graph of the Moxel DAG sizes at different resolutions as a percentage of the corresponding SVO.

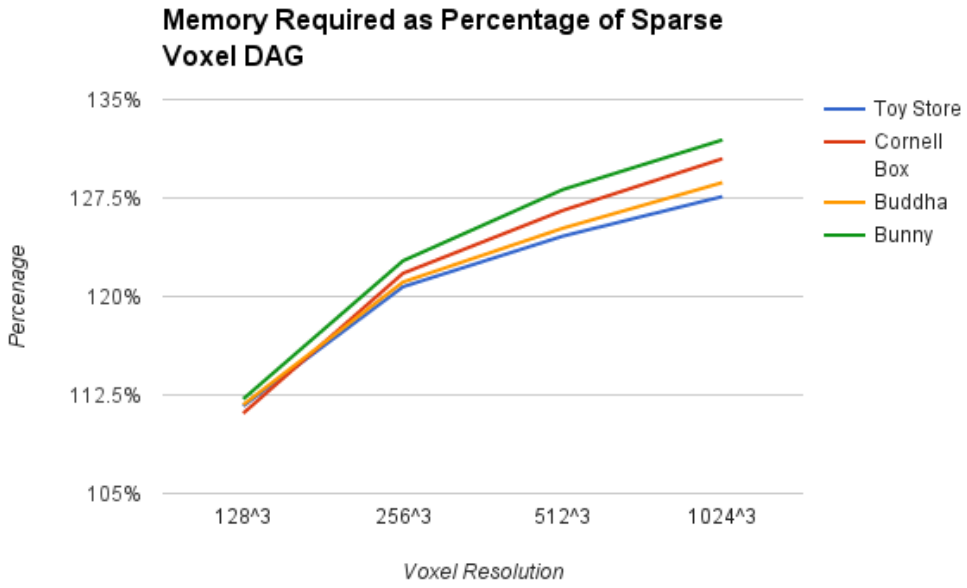


Figure 5.3: A graph of the Moxel DAG sizes at different resolutions as a percentage of the corresponding Sparse Voxel DAG.

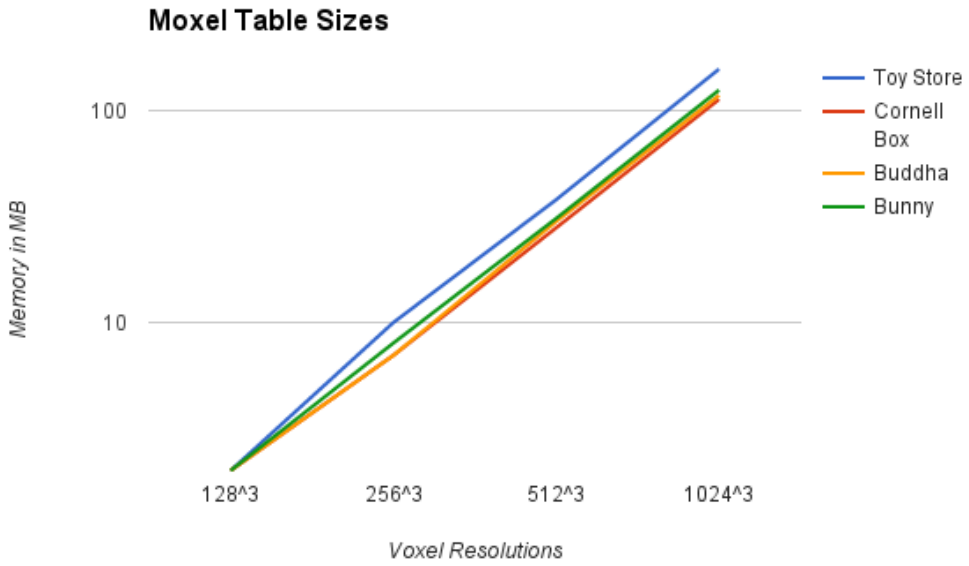


Figure 5.4: A graph of the Moxel Table sizes at different resolutions. Note that the left vertical axis uses a logarithmic scale.

the resolution is increased by. Note that whether it is a Moxel DAG or an SVO, both will need to include this exact same material information that is stored in the Moxel Table.

5.4.2 Build Times

Our results for the build times can be seen in Table 5.4. Note that voxelization and the building of the SVO need to be done to create the Sparse Voxel DAG, the Moxel DAG, and the Moxel Table. The times for voxelization and building the SVO are not included in the build times for the Sparse Voxel DAG, the Moxel DAG, and the Moxel Table in Table 5.4.

A large percent of the total build time is spent in voxelization. Due to time constraints, a simple parallel method of voxelization was implemented. Specifically, for each triangle in the mesh, its bounding box was used to calculate which voxels were possible for it to intersect with,

		Build Times in ms		
Scene		256^3	512^3	1024^3
ToyStore	Voxelization	16,446	43,401	232,131
	SVO	2	11	152
	Sparse Voxel DAG	452	5,026	64,915
	Moxel DAG	917	9,223	102,978
	Moxel Table	2,508	19,960	158,072
CornellBox	Voxelization	3821	12,483	57,117
	SVO	1	12	187
	Sparse Voxel DAG	355	3,422	43,173
	Moxel DAG	826	8,804	88,508
	Moxel Table	3,127	24,479	199,107
Buddha	Voxelization	3,563	11,017	53,243
	SVO	1	13	143
	Sparse Voxel DAG	235	2,434	32,916
	Moxel DAG	759	7,735	77,546
	Moxel Table	2,993	24,759	204,640
Bunny	Voxelization	2,912	33,288	50,668
	SVO	2	14	137
	Sparse Voxel DAG	298	3,231	39,553
	Moxel DAG	872	8,648	85,329
	Moxel Table	3,067	24,057	205,595

Table 5.4: Timing results for each part in building Moxel DAGs, and Sparse Voxel DAGs.These include voxelization, SVO building, making the Sparse Voxel DAG from the SVO,making the Moxel DAG from the SVO, and building the Moxel Table. Making the SparseVoxel DAG, the Moxel DAG, and the Moxel Table do not include building the SVO or Voxelization in their times. Voxel resolutions are listed in the second row and result from8, 9, and 10 level graphs respectively. Each of the build times is an average of three runs.

and then each of those voxels was tested. This was parallelized with each unit of work being the calculations done for each triangle. For a faster method of voxelization, see [Schwarz and Seidel, 2010].

Building the SVO took very little of the total build time. This is due to its simple structure and that only one pass over the entire tree is ever needed.

When comparing the times needed to build a Sparse Voxel DAG from an SVO, and a Moxel DAG from an SVO, the Moxel DAG takes significantly longer. On average it took about two times as long. This is due to the need to traverse the entire Moxel DAG an extra time to calculate the empty counts.

If the Moxel DAG's build time included the Moxel Table's build time, its performance is even worse compared to the Sparse Voxel DAG. The long build times for the Moxel Tables are due to it being written out in Morton order by going voxel by voxel. Each voxel in the scene is checked to see if is full, and if so its material information is written to the next available spot in the Moxel Table.

		Ray Tracing Times in ms		
Scene		256^3	512^3	1024^3
ToyStore	Sparse Voxel DAG	14,556	14,422	15,577
	Moxel DAG	25,045	24,032	29,505
CornellBox	Sparse Voxel DAG	8,685	8,952	8428
	Moxel DAG	14,668	14,225	15,512
Buddha	Sparse Voxel DAG	8,602	8,845	8,458
	Moxel DAG	14,210	13,824	15,131
Bunny	Sparse Voxel DAG	9,457	9,518	11,218
	Moxel DAG	16,642	15,694	15,985

Table 5.5: Comparison of ray tracing times for a Sparse Voxel DAG Raytracer, and a Moxel DAG Raytracer for each of the test scenes. Voxel resolutions are listed in the second row and result from 8, 9, and 10 level graphs respectively.

5.4.3 Rendering Times

Rendering times were recorded for a Sparse Voxel DAG ray tracer and a Moxel DAG ray tracer, for our test scenes at three different voxel resolutions. These can be seen in Table 5.5.

On average, the Moxel DAG takes 1.7 times as long to render as the Sparse Voxel DAG. This is due to the increased amount of calculations needed to obtain the material information.

Because material information cannot be saved in the Sparse Voxel DAG, the normal is calculated based on which side of the voxel it hits, and the rest of the material information is hard coded as one set of values for the entire scene. For the Moxel DAG, to get the material information, extra calculations have to be done to determine its index in the Moxel table during traversal. These calculations are executed on every traversal no matter if that branch eventually turns out to be an intersection or not.

5.4.4 Limitations

Like many voxel data structures, one of the main limitations of this algorithm is its memory requirements. While the memory needed to store the geometric information in Moxel DAGs has been shown to be a fraction of that of SVOs, the material information, which both need, has not been reduced at all. This makes Moxel DAGs practically unusable at very high resolutions.

While the memory savings of Moxel DAGs could allow higher resolution voxel data to be rendered than traditional SVOs, it does not do so well enough to mitigate the blockiness that is known to plague renders of voxel data structures. This can be seen in Figure 5.5.

As written, when the Moxel Table is constructed, per vertex attributes like normals and texture coordinates are not interpolated. This prevents us from implementing many common

a) Moxel DAG Render at 1024^3

b) Triangle Render

Figure 5.5: An example of the blockiness when the resolution of the voxels is not high enough. Here we compare a Moxel DAG rendered at 10243
in a and a triangle version of

algorithms used for visually rich rendering such as smooth shading, texturing, and normal mapping.

Lastly, with the Moxel DAG taking an average 1.7 times as long to ray trace as the Sparse Voxel DAG, the speed of rendering could be considered a limitation. Note that there are differences between rendering a Sparse Voxel DAG and a Moxel DAG. Using a Moxel DAG allows you to have multiple materials for different objects in the scene, while using just a Sparse Voxel DAG does not. In the original author's implementation, to create the images for the paper, a traditional SVO had to be kept in memory along with the Sparse Voxel DAG to store material information [Kampe et. al, 2013].

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

During the development of the this work, some possible areas for improvement, and further research were noted. In this section we will conclude our work, and list areas that we suggest being explored.

6.1 Conclusion

In this thesis, we discussed the current state of voxel data structures, and presented Moxel DAGs, an extension to Sparse Voxel DAGs [Kampe et. al, 2013] that allows for the addition of material information. While our method does require some overhead in memory and speed over Sparse Voxel DAGs, it allows for more visually rich rendering, and is much more memory efficient than traditional SVOs. Because it also stores the equivalent information as SVOs, it has the exact same visual quality at the same resolutions. Due to its memory requirements, and ray tracing performance, a possible application of our work would be as the geometric data structure for static elements used when rendering visually rich scenes in movies or video games.

The demand for more realistic and better quality images will only increase as time goes on. Material information is a key component in how we render high quality images. Storing and accessing material information for voxel data structures is currently pushing our computational resources to their limits, but new advances are happening every day. One can only imagine what the future will hold, but the gap between what is real and what is rendered is only getting smaller.

6.2 Future Work

During the construction of the Moxel Table, per vertex attributes like normals and texture coordinates were not interpolated. This is something that could be added to allow for effects such as smooth meshes, textured surfaces, and normal mapped surfaces. One possible way this can be done is, given a voxel and the triangle that the voxel was created from, we could find the closest point on the triangle to the center of the voxel, that is still within the voxel, and use that point as the sample point for interpolating per vertex attributes.

Of the Moxel DAG, the Moxel Table, and Material Table, our results have shown that the Moxel Table takes up the vast majority of the space. Decreasing its memory requirements would allow for Moxel DAGs of much higher resolutions to be used. One possible way to do so would be to use the methods introduced in [Laine and Karras, 2013] to compress material information.

To allow for the use of level of detail while ray tracing, the material information of child nodes could be averaged and stored in the parent nodes. This would allow the ray tracer to stop early at a level in the DAG if the size of the voxel at that current level is smaller than the size of the pixel that is being rendered. This could potentially speed up the ray tracing of the Moxel DAG.

Other than voxelization, building the Moxel Table is one of the slowest portions of the code. The method as presented cannot be parallelized. One possible way of making this parallel is splitting the Moxel DAG up into as many parts as the number of threads available. Then, each thread can traverse its portion of the Moxel DAG, in morton order, calculating its Moxel Index as it goes. For every filled voxel, its material information is calculated and written to its place in the Moxel Table.

A considerable limitation of our work is its rendering performance. It has been shown that Moxel DAGs take 1.7 times as long to render as Sparse Voxel DAGs. Using methods like the packet traversal algorithm described in [Knoll et. all, 2009], or doing more in depth work parallelizing and vectorizing the implementation may help increase its performance.

Without including material information, the difference in memory size between Sparse Voxel DAGs and Moxel DAGs are entirely due to the addition of empty counts to their nodes. In all of our tests most of the voxels in the scene are empty. One possible way of saving more space in Moxel DAGs is to keep the count of the number of filled voxels instead of the number of empty voxels. Because these numbers would be smaller than the empty counts, they could be stored in less bits and still allow for the calculation of Moxel indexes.

Another possible method of storing and accessing material information for a Sparse Voxel DAG is to keep a hash table of material information similar to our proposed Moxel Table. The voxel index then could be calculated upon traversal, and used as the key to the hash table. This would enable the access of material information without necessitating additional memory to store the empty counts in the DAG.

Finally, another alternative for storing materials is to save them in the leaves of the Moxel DAG. Each unique leaf would then store the materials for every voxel that leaf represents. This could potentially increase caching, and lead to better ray tracing performance.

BIBLIOGRAPHY

[Amanatides and Woo, 1987] Amanatides, J. and Woo, A. (1987). A fast voxel traversal algorithm for ray tracing. InIn Eurographics 87, pages 3–10.

[Appel, 1968] Appel, A. (1968). Some techniques for shading machine renderings of solids. In Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring), pages 37–45, New York, NY, USA. ACM.

[Crassin et al., 2009] Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. (2009). Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In Proceed-ings of the 2009 Symposium on Interactive 3D Graphics and Games, I3D '09, pages 15–22, New York, NY, USA. ACM.

[Gobbetti and Marton, 2005] Gobbetti, E. and Marton, F. (2005). Far voxels: A multi-resolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. InACM SIGGRAPH 2005 Papers, SIGGRAPH '05, pages 878–885, New York, NY, USA. ACM.

[Huang et al., 2006] Huang, Y., Peng, J., Kuo, C.-C. J., and Gopi, M. (2006). Octree-based progressive geometry coding of point clouds. InProceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics, SPBG'06, pages 103–110, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Kampe et al., 2013] Kampe, V., Sintorn, E., and Assarsson, U. (2013). High resolution sparse voxel dags. ACM Trans. Graph., 32(4):101:1–101:13.

[Katajainen and Makinen, 1990] Katajainen, J. and Makinen, E. (1990). Tree compression and optimization with applications.International Journal of Foundations of Computer Science, 1(04):425–447.

[Knoll et al., 2006] Knoll, A., Wald, I., Parker, S., and Hansen, C. (2006). Interactive isosurface ray tracing of large octree volumes. InIn Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pages 115–124.

[Knoll et al., 2009] Knoll, A. M., Wald, I., and Hansen, C. D. (2009). Coherent multiresolution isosurface ray tracing.Vis. Comput., 25(3):209–225.

[Laine and Karras, 2010] Laine, S. and Karras, T. (2010). Efficient sparse voxel octrees – analysis, extensions, and implementation.

[Museth, 2013] Museth, K. (2013). Vdb: High-resolution sparse volumes with dynamic topology. ACM Trans. Graph., 32(3):27:1–27:22.

[Parker and Udeshi, 2003] Parker, E. and Udeshi, T. (2003). Exploiting self-similarity in geometry for voxel based solid modeling. InProceedings of the Eighth ACM Symposium on Solid Modeling and Applications, SM '03, pages 157–166, New York, NY, USA. ACM.

[Phong, 1973] Phong, B. T. (1973). Illumination for Computer-generated Images. PhD thesis. AAI7402100.

[Revelles et al., 2000] Revelles, J., Urea, C., and Lastra, M. (2000). An efficient parametric algorithm for octree traversal. InJournal of WSCG, pages 212–219.

[Schwarz and Seidel, 2010] Schwarz, M. and Seidel, H.-P. (2010). Fast parallel surface and solid voxelization on gpus. InACM SIGGRAPH Asia 2010 Papers, SIGGRAPH ASIA'10, pages 179:1–179:10, New York, NY, USA. ACM.

[Webber and Dillencourt, 1988] Webber, R. and Dillencourt, B. (1988). Compressing quadtrees via common subtree merging. (no. 2137).

[Whitted, 2005] Whitted, T. (2005). An improved illumination model for shaded display. In ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, New York, NY, USA. ACM.