EXPLORING MATERIAL REPRESENTATIONS FOR 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

Steven Pineda

June 2021

COMMITTEE MEMBERSHIP

TITLE: Exploring Material Representations for Sparse Voxel DAGs

AUTHOR: Steven Pineda

DATE SUBMITTED: June 2021

COMMITTEE CHAIR: Zoë Wood, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Aaron Keen, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Theresa Migler, Ph.D.
Professor of Computer Science

ABSTRACT

Exploring Material Representations for Sparse Voxel DAGs

Steven Pineda

Ray tracing is a popular technique used in movies and video games to create compelling visuals. Ray traced computer images are increasingly becoming more realistic and almost indistinguishable from real-word images. Due to the complexity of scenes and the desire for high resolution images, ray tracing can become very expensive in terms of computation and memory. To address these concerns, researchers have examined data structures to efficiently store geometric and material information. Sparse voxel octrees (SVOs) and directed acyclic graphs (DAGs) have proven to be successful geometric data structures for reducing memory requirements. Moxel DAGs connect material properties to these geometric data structures, but experience limitations related to memory, build times, and render times. This thesis examines the efficacy of connecting an alternative material data structure to existing geometric representations.

The contributions of this thesis include the creation of a new material representation using hashing to accompany DAGs, a method to calculate surface normals using neighboring voxel data, and a demonstration and validation that DAGs can be used to super sample based on proximity. This thesis also validates the visual acuity from these methods via a user survey comparing different output images. In comparison to the Moxel DAG implementation, this work increases render time, but reduces build times and memory, and improves the visual quality of output images.

ACKNOWLEDGMENTS

Thanks to:

- My parents, Loren and Vicki, for always supporting me and celebrating my achievements
- My brother, Ryan, for introducing me to Cal Poly and being such a great role model
- Zoë Wood, for being the best advisor and pushing me to discover my own potential
- Aaron Keen, for answering all my questions throughout four courses and helping me realize I could succeed in graduate school
- Theresa Migler, for bringing so much positivity into all of our encounters and inspiring empathy and kindness
- The graphics group, for building an encouraging and productive community
- All of my other family and friends, for giving me breaks from school to play sports, eat good food, relax, laugh, and smile
- Brent Williams, for providing base code that I could learn from, experiment with, and extend
- Andrew Guenther, for uploading this template

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Ray tracing is popular in industries like movies and video games due to its ability to simulate realistic lighting and create visually attractive images. It enables computer-generated scenes and environments to be rendered with real-world effects like shadows and reflections. Unfortunately, this rendering technique performs many computations and uses a lot of memory. As scenes gain more geometry and materials, computational and memory limitations become more apparent and grow in severity. Consequently, a lot of computer graphics research focuses on how to improve efficiency.

1.1 Ray Tracing

Ray tracing is a rendering technique used to create 2D images of 3D worlds [15]. In this technique, rays are shot from a virtual camera through an image plane and intersected with scene geometry to determine lighting. Ray tracing is computationally intensive because at least one ray is shot through each pixel in the image plane. The color of each pixel depends on several factors like which objects are hit, what materials are associated with the hit objects, what directions the hit surfaces are facing, and where the light sources in the scene are located. Figure 1.1 depicts a few steps of the ray tracing algorithm.

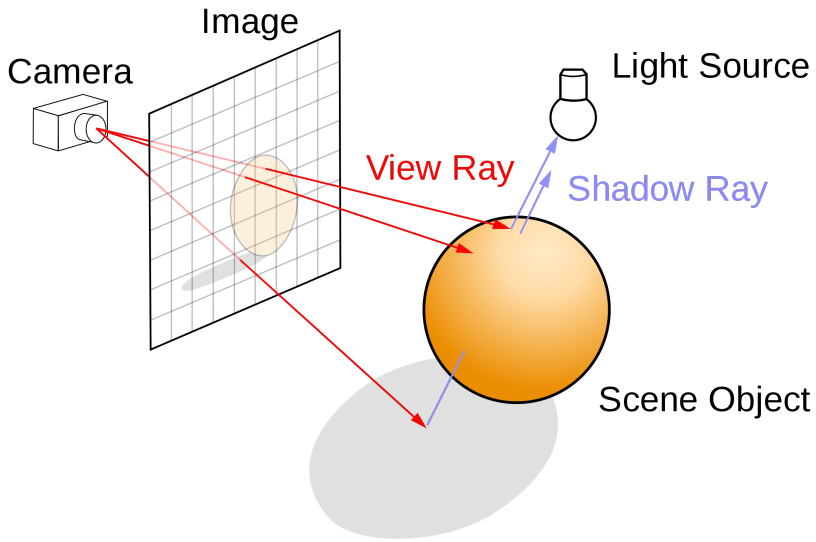


Figure 1.1: A diagram of the ray tracing process, which produces a 2D image of a 3D scene containing a sphere. By Henrik [16] and licensed under CC BY-SA 4.0.

1.2 Geometry

One of the first steps of a ray tracing program is to read in and process the geometry in a scene. The surface of geometric objects can be approximated by a collection of shapes, like triangles. Figure 1.2 shows a bunny triangular mesh in Blender. Mesh files, like the OBJ files used in this thesis, store vertex and triangle connectivity information about 3D models. OBJ files can also reference other files with material information.

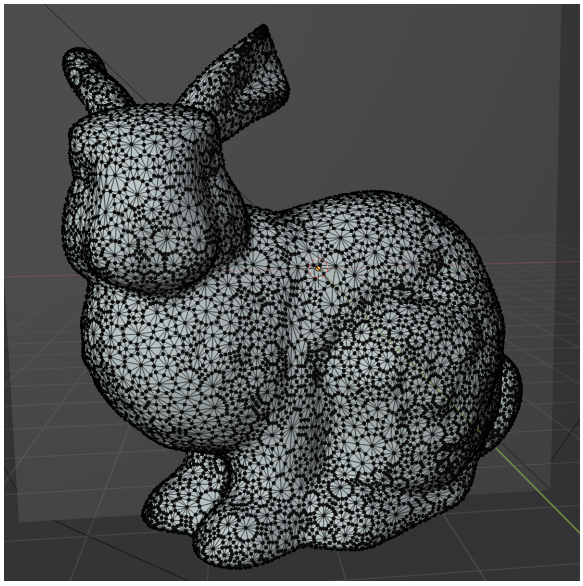


Figure 1.2: The surface of a bunny approximated by small triangles.

1.3 Materials

Material properties determine how light interacts with objects and affect how objects appear. Figure 1.3 shows a model rendered with different materials. Materials specify things like the color and smoothness of objects. They can model a wide range of surfaces such as emerald, plastic, rubber, and glass. Like geometric information, material information is stored in a file that needs to be parsed before ray tracing a scene. This thesis uses MTL files to store different materials. Each triangle in the OBJ files has a material in an MTL file associated with it.

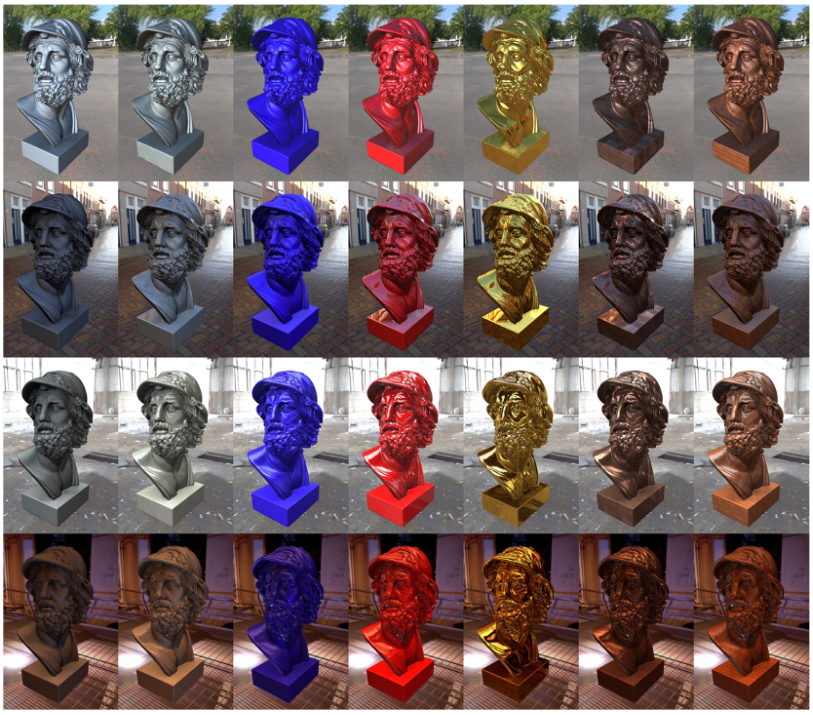


Figure 1.3: A model rendered with different materials [4].

1.4 Moxel DAGs

This thesis is largely inspired by Brent Williams's work with Moxel DAGs [27]. Williams created a method to connect material information to an efficient geometric data structure, High Resolution Sparse Voxel DAGs [17]. His method allows the successful rendering of images with material information, but experiences some memory and performance limitations. The work in this thesis extends and modifies the Moxel DAG implementation and addresses some areas that Williams recommended for future work.

1.5 Contributions

This thesis presents an exploration of rendering with the ultimate goal of improving ray tracing efficiency in terms of computation and memory. This work builds on previously explored geometric data structures, SVOs and DAGs, and examines how to effectively store and utilize materials and scene geometry. The main contributions of this work are listed below:

- Creation of a new material representation based on hashing to connect to DAGs
- A method to calculate surface normals so that normals do not need to be stored in the material data structure
- Demonstration and validation that DAGs can be used to super sample based on proximity
- Validation of visual quality via a user survey comparing output images of different algorithms used throughout this thesis

Chapter 2

BACKGROUND

This thesis focuses on efficiently representing and storing geometric and material information for ray tracing. It is useful to be familiar with some terminology related to these concepts. This section provides background on the bidirectional reflectance distribution function (BRDF), voxelization, spatial data structures, and Morton coding.

2.1 BRDF

The bidirectional reflectance distribution function (BRDF) defines how light is reflected at a given surface in a certain direction. Wynn [28] defines a BRDF as "a function of incoming (light) direction and outgoing (view) direction relative to a local orientation at the light interaction point." A surface's orientation is typically represented by a normal vector, which is a vector that is perpendicular to the surface. Figure 2.1 shows a surface, S, and several normal vectors throughout the surface. BRDFs are relevant to computer graphics because they provide a way to realistically light and render scenes. This work uses the Phong Reflection Model.

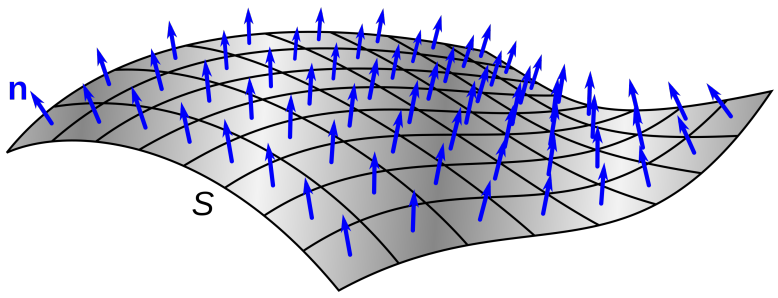


Figure 2.1: Normal vectors along the curved surface, S. By Chetvorno [5].

2.1.1 Phong Reflection Model

The Phong Reflection Model is a special case of a BRDF and is used to locally approximate how light interacts with a surface. This model is used to calculate the color of object surfaces that reflect into a viewer's eye. The reflected color can be decomposed into three components: ambient, diffuse, and specular. Figure 2.2 depicts the contributions of each component to a final output image. The final reflected color can be described by the following equation:

$$c_{reflected} = c_{ambient} + c_{diffuse} + c_{specular}$$

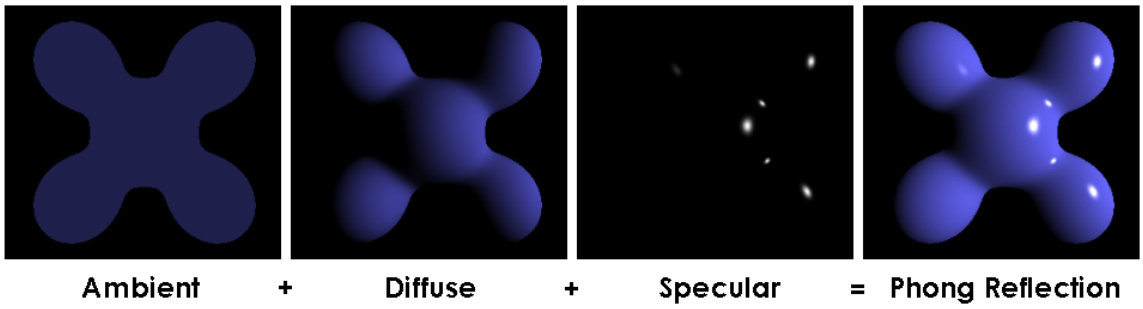


Figure 2.2: The ambient, diffuse, and specular component contributions to a final image rendered with the Phong Reflection Model. By Brad Smith [23] and licensed under CC BY-SA 3.0.

Ambient reflection refers to the small amount of illumination throughout scenes that occurs due to light bouncing everywhere. The ambient reflection at a given surface point depends on the intensity of the ambient light and the surface material. The ambient component can be calculated by the following equation:

$$c_{ambient} = I_a * K_a,$$

where I_a is the ambient light intensity and K_a is the ambient reflection coefficient. K_a is a material property of the object being illuminated. It has a red, green, and blue component, which are floats in the range [0.0, 1.0]. Generally, the ambient reflection color can be thought of as the color an object would be in a shadow.

Diffuse reflection represents light that is uniformly scattered in all directions on matte surfaces. More light gets reflected when objects are oriented toward the light source. In other words, the diffuse component at a particular point increases as the normal of a surface, N, aligns with the light vector, L. L is the direction from the surface point to the light source. Figure 2.3 shows examples of the N and L vectors for a particular scenario. The dot product operator can be used to quantify how close two vectors align. The dot product of the normalized normal vector, $\hat{N}$, and the normalized light vector, $\hat{L}$, will be positive when they are in the same direction, 0 when they are perpendicular, and negative when they are in opposite directions. Figure 2.2 illustrates how the diffuse component is bigger and contributes more light for regions on the surface directed toward the light. The diffuse component can be calculated by the following equation:

$$c_{diffuse} = \sum_{all\ lights} I_d * K_d * (\hat{L} \cdot \hat{N}),$$

where I_d is the diffuse light intensity, K_d is the diffuse reflection coefficient, $\hat{L}$ is the normalized light vector, and $\hat{N}$ is the normalized normal vector at the surface. Note that this equation has a summation over all lights because each light in a scene contributes to the diffuse reflection at a certain point. Similar to K_a, K_d is a material property of the surface. Each has a red, green, and blue component.

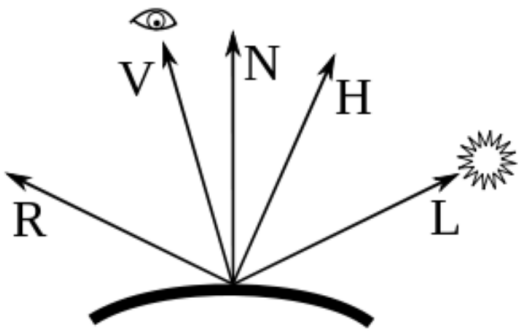


Figure 2.3: A visualization of the reflected vector, R, the view vector, V, the normal vector, N, the half vector, H, and the light vector, L. By Ian Dunn [10].

Specular reflection models shiny, smooth surfaces and produces specular highlights on surfaces. Light bounces off specular surfaces like a mirror so specular reflection is strongest along the direction of the reflected light vector, R. The view vector, V, is the direction from the surface point to the viewer. Figure 2.3 shows what these vectors look like for a particular scenario. The specular component is maximized when $R = V$. Its equation is

$$c_{specular} = \sum_{all\ lights} I_s * K_s * (\hat{V} \cdot \hat{R})^{\alpha},$$

where I_s is the specular light intensity, K_s is the specular reflection coefficient, $\hat{V}$ is the normalized view vector, $\hat{R}$ is the light's reflected vector along the surface normal, and α is a shininess coefficient that controls the sharpness and shininess of the specular highlights. Both K_s and α are properties of a surface's material.

After substituting the equations for each component, the entire Phong Reflection Model is denoted by the following equation:

$$c_{reflected} = I_a * K_a + \sum_{all\ lights} (I_d * K_d * (\hat{L} \cdot \hat{N})) + (I_s * K_s * (\hat{V} \cdot \hat{R})^{\alpha})$$

The Phong Reflection Model is described to familiarize readers with and build an understanding of the values associated with materials. In the equation, K_a, K_d, K_s, and α are constants used to represent a specific material. Figure 2.4 shows the K_a, K_d, K_s, and α values for a brass material.

```
//Brass
float[] mat_ambient ={ 0.329412f, 0.223529f, 0.027451f,1.0f };
float[] mat_diffuse ={ 0.780392f, 0.568627f, 0.113725f, 1.0f };
float[] mat_specular ={ 0.992157f, 0.941176f, 0.807843f, 1.0f };
float shine = 27.8974f;
```

Figure 2.4: The shine constant and the ambient, diffuse, and specular reflection coefficients for a brass material. By Ostfold University College Department for Information Technology [22].

2.2 Voxelization

A voxel, or volume element, is a single unit in a 3D grid and can be used to visualize volumetric data. It can be thought of as the 3D equivalent of a 2D pixel. Voxels are a useful generic representation for geometry because they break up complex objects into small building blocks. Voxel-based rendering also has great potential to efficiently render large and detailed scenes since high-resolution voxel data can be stored compactly and hierarchically as sparse voxel octrees (SVOs) [3].

Voxelization is the process of converting models into individual voxels that approximate the objects. To voxelize a mesh, each triangle in the mesh is voxelized. One way to voxelize a triangle is to do the following [27]:

- Calculate the bounding box around a given triangle
- Calculate the minimum and maximum x, y, and z indices into the 3D voxel grid from the triangle's bounding box

- Use the minimum and maximum x, y, and z voxel indices to loop through a subset of the voxels in the 3D voxel grid
- Test if the triangle intersects with each voxel and mark the voxel as occupied if an intersection exists

When an intersection occurs between a triangle and a voxel, the corresponding cell in a 3D voxel grid can be updated to store useful information. For example, the entire base primitive (i.e., the actual triangle) and its material properties can be associated with an occupied cell. Other methods may store the triangle's normal coordinates and material properties, but not the geometric representation of the triangle; these methods use the voxel as the base primitive instead. The size of a voxel unit in a 3D voxel grid determines how detailed and accurate object approximations will be. Smaller voxels produce better approximations but require more memory since the number of voxels in the 3D grid increases. Figure 2.5 shows the result of voxelizing a triangle at different resolutions. As the voxel size decreases, the edges of the triangle become smoother.

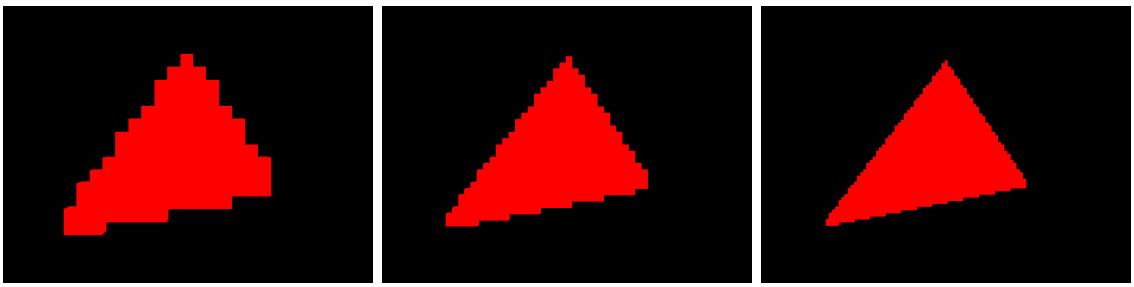


Figure 2.5: Triangle voxelization with varying voxel sizes. The voxel size decreases from left to right.

2.3 Spatial Data Structures

During the ray tracing process, it is necessary to determine if each ray hits an object in the scene. One way to do this is to loop through each object for a given ray and perform a ray-object intersection check. However, this process can become very time consuming as the number of objects increases. Spatial data structures attempt to optimize the process of finding ray-object intersections by organizing objects based on their locations in the scene. By grouping objects based on location, spatial data structures contain information on which objects are far away from a given ray. Thus, some objects can be skipped entirely when checking for intersections. Two common spatial data structures are bounding volume hierarchies and octrees.

2.3.1 Bounding Volume Hierarchies

A bounding volume hierarchy (BVH) encloses the scene geometry in a series of nested volumes [11]. It can be represented as a binary tree of bounding boxes, with each bounding box enclosing all of the geometry beneath it. Each node stores left and right pointers to nodes representing smaller regions of space. The leaf nodes of this tree are individual objects and the root node is a bounding box containing all the objects. A BVH can be constructed by sorting the objects by their center coordinates and then splitting the object list into two groups. Each group is assigned to the left or right attribute of a node. The subdivision repeats until each object is a leaf in the tree. Figure 2.6 shows how a scene can be represented with a BVH. Ray tracing a BVH can save time because the data structure prevents further traversal in parts of the tree if a ray does not intersect with the bounding box encapsulating that subtree.

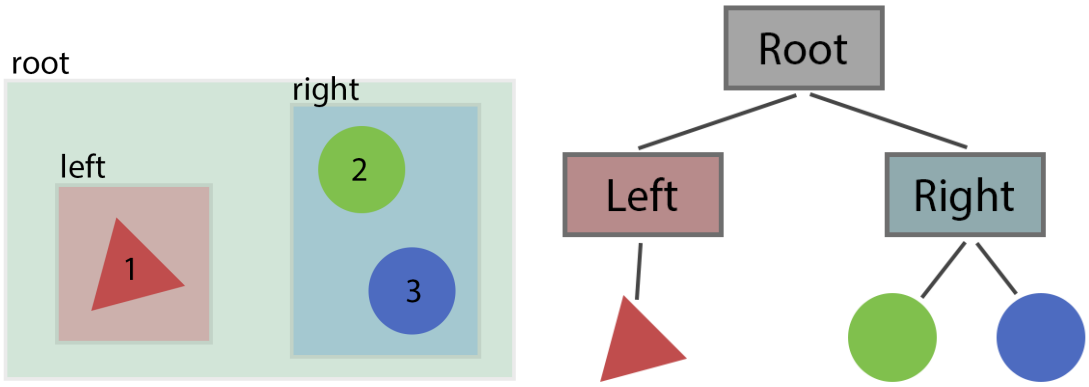


Figure 2.6: A scene with bounding volumes (left) and a bounding volume hierarchy graph for that scene (right). By Ian Dunn and Zoë Wood [11].

2.3.2 Octrees

An octree is another data structure that allows spatial queries. It is a tree in which each internal node has eight children. Like bounding volume hierarchies, octrees have nodes that correspond with specific locations in a scene. Each node represents a cube that bounds all the geometry beneath it. To produce eight children, a node is subdivided into eight equal-sized cubes. Figure 2.7 depicts the subdivision of certain nodes in an octree. Octrees can be used to represent 3D voxel grids by having each leaf node correspond to one voxel in the grid. During ray tracing, a ray is first tested for intersection with the root node's bounding box. If there is no intersection, then the ray does not intersect with any of the voxels beneath the root node. If there is an intersection, then intersections are recursively checked on the child nodes.

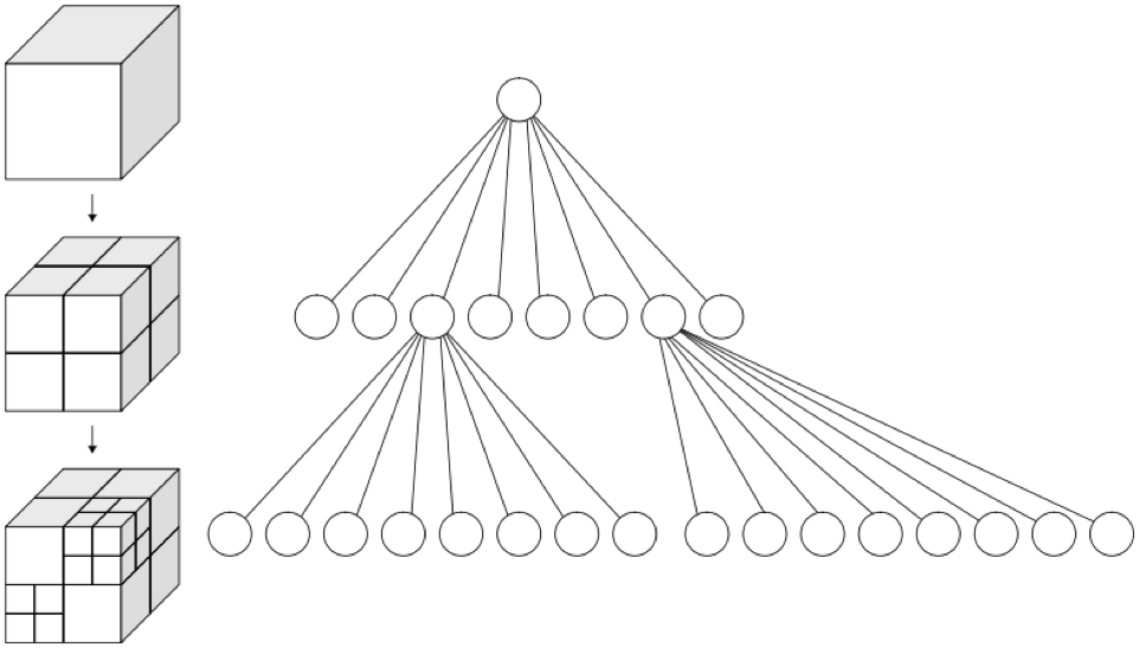


Figure 2.7: Octree subdivision (left) and the corresponding octree (right). By WhiteTimberwolf [26] and licensed under CC BY-SA 3.0.

2.4 Morton Codes

Morton encoding is a technique that maps multidimensional data into one dimension while preserving data locality. With this method, 2D and 3D coordinates can be converted into a single number. Furthermore, nearby coordinates' Morton codes will be close in value to each other. Figure 2.8 shows the process of constructing the Morton codes from 2D points. Note that this process can be generalized to any dimension of data. Morton codes are created by converting each coordinate to binary and interleaving the bits. Morton order is sometimes referred to as Z-order because the ordering of the data points makes a Z pattern.

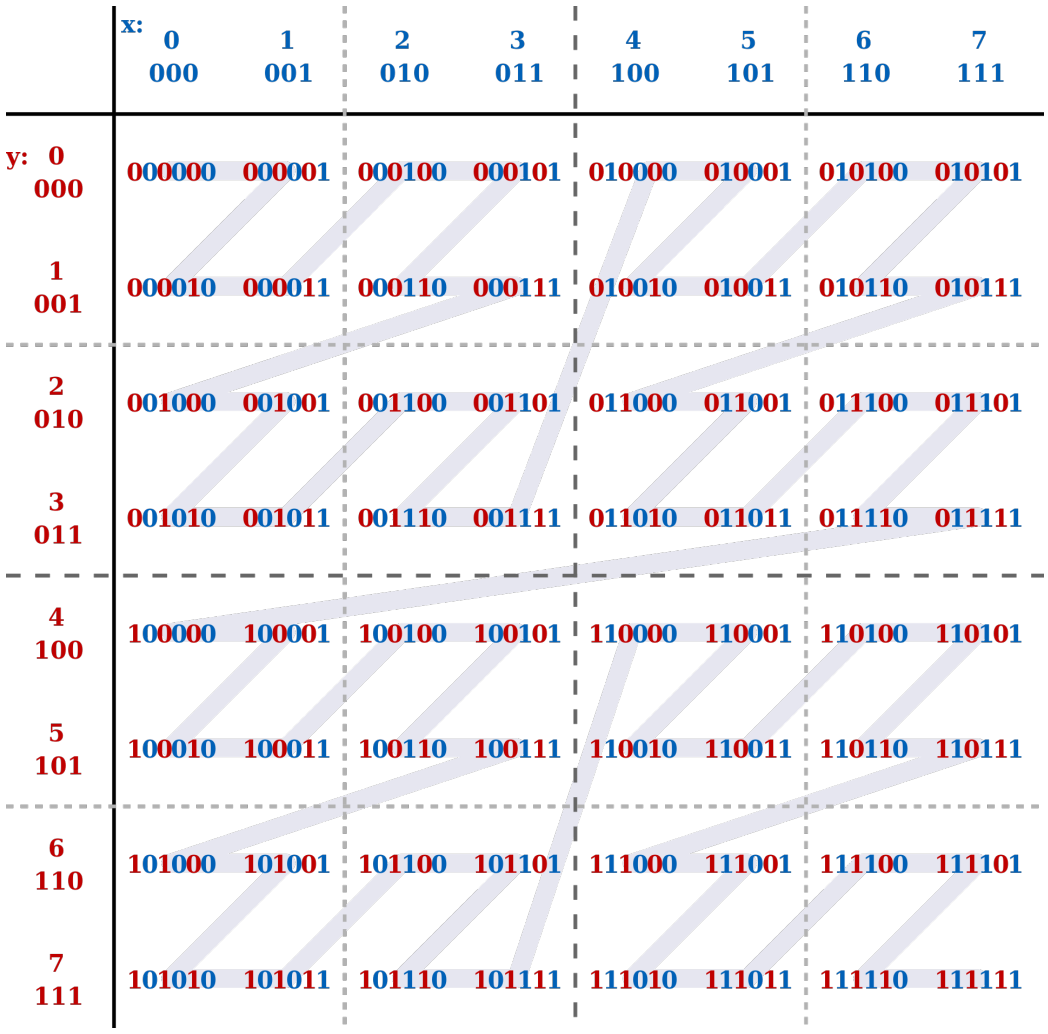


Figure 2.8: The Morton codes for 2D coordinates. Incrementally following the produced Morton codes creates a Z pattern. By David Eppstein [13].

Morton codes are relevant to this thesis because they can be used to identify voxels in a 3D grid. They also aid in constructing spatial data structures like octrees because they provide an explicit ordering of the child nodes. Figure 2.9 depicts a quadtree representing a 2D grid. The leaf nodes are in Morton order and each leaf corresponds to a cell in the 2D grid. Due to the spatial locality provided by Morton coding, nearby cells have similar Morton codes and are therefore located closely in the tree.

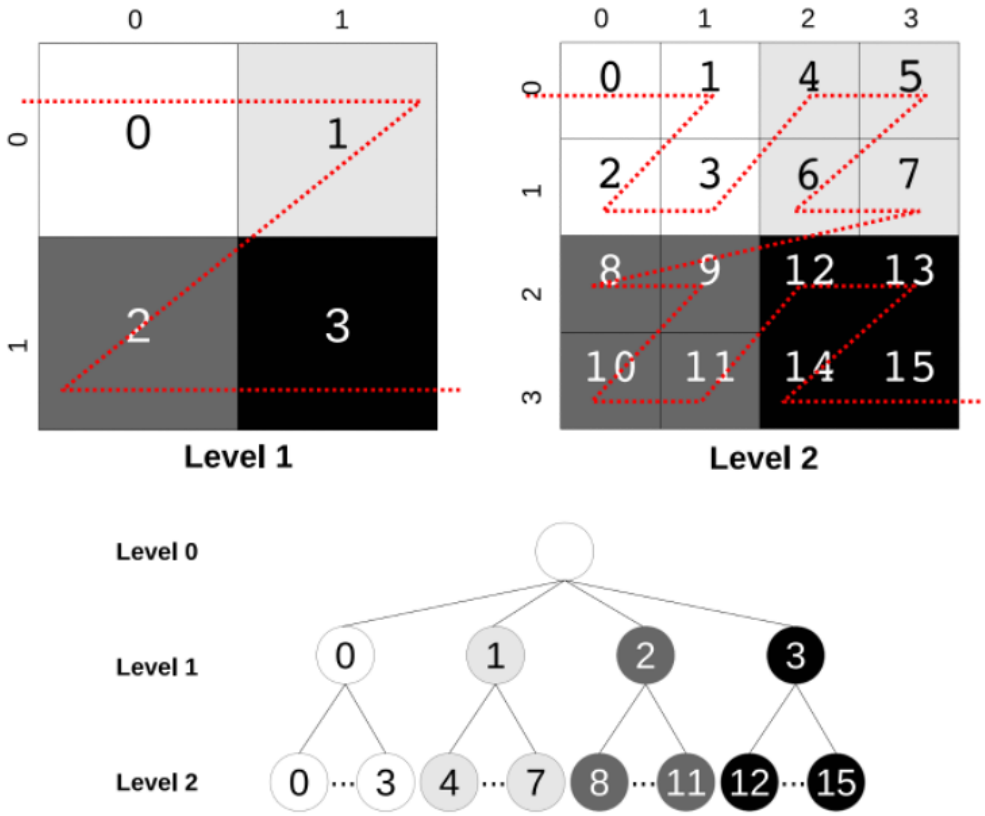


Figure 2.9: A quadtree representing a 2D grid. The leaf nodes follow Morton order and correspond to the grid cells. By Baert et al. [3].

Chapter 3

RELATED WORK

A lot of ray tracing research goes into examining useful data structures because data structures are important to decrease memory and render times. This section presents relevant research on octrees, Sparse Voxel DAGs, Moxel DAGs, material data structures, and compression techniques. High Resolution Sparse Voxel DAGs [17] and Moxel DAGs [27] are described in a little more detail since this work primarily builds off them.

3.1 Octrees

An octree is a 3D hierarchical spatial data structure containing nodes that can be subdivided into eight equal and smaller regions. Each internal node has a pointer to each of its eight children. In "Efficient Processing of Large 3D Point Clouds," Elseberg et al. [12] implemented an efficient octree capable of storing one billion data points collected by autonomous robots with laser scanners. They used an octree because they wanted a structure that stores raw data, has fast access, is memory efficient, and allows efficient ray tracing.

3.1.1 Sparse Voxel Octrees (SVOs)

Sparse Voxel Octrees (SVOs), a popular extension of octrees, are memory efficient because they do not need to encode empty regions of space. Sparsity can be achieved by setting child pointers to null when the child nodes represent empty space. Null

child pointers prevent further subdivision down empty routes. Kämpe et al. [17] show that nodes can alternatively be implemented with a childmask and pointers to the non-empty children. An SVO node's childmask is eight bits where bit i is set if child i contains geometry. The use of childmasks eliminates the need to store unused child pointers.

Baert et al. [3] discovered an out-of-core algorithm to convert a triangle mesh into an intermediate high-resolution 3D voxel grid. They used this voxel grid to construct an SVO. Their algorithm handles "extremely large" triangle meshes and uses less memory than in-core algorithms. Laine and Karras [19] created an efficient sparse voxel octree (ESVO) to represent complex scenes on modern GPUs. They store contour data in each voxel to help approximate the geometry. The contour data is a pair of parallel planes matching the orientation of the surface. This additional data allows better rendering performance because traversals of their data structure do not need to go as deep in the tree if the current voxel's contour provides a sufficient approximation. Crassin et al. [6] use dynamic SVOs on the GPU to render large volumetric data sets. Their tree structure stores a constant value or a pointer to a brick, which is a small voxel grid that approximates the original volume at that node. The authors state that their method achieves "interactive to real-time rendering performance for several billion voxels."

3.2 High Resolution Sparse Voxel DAGs

One of the core papers extended in this thesis is "High Resolution Sparse Voxel DAGs" by Kämpe et al. [17]. Kämpe et al. first encode an N^3 binary voxel grid as an SVO. N is the number of voxels that make up the width of the voxel grid. A binary voxel grid is a structure in which each cell is represented by one bit; the bit

is 0 if the voxel is empty or 1 if the voxel contains geometry. An SVO can represent an N^3 voxel grid by recursively dividing an octree L times, where $N = 2^L$. L, the max level of the tree, is the number of levels below the root node. Each leaf node corresponds to one voxel in the 3D grid. Figure 3.1 shows an N^3 voxel grid where $N = 4$ and the corresponding octree where $L = 2$.

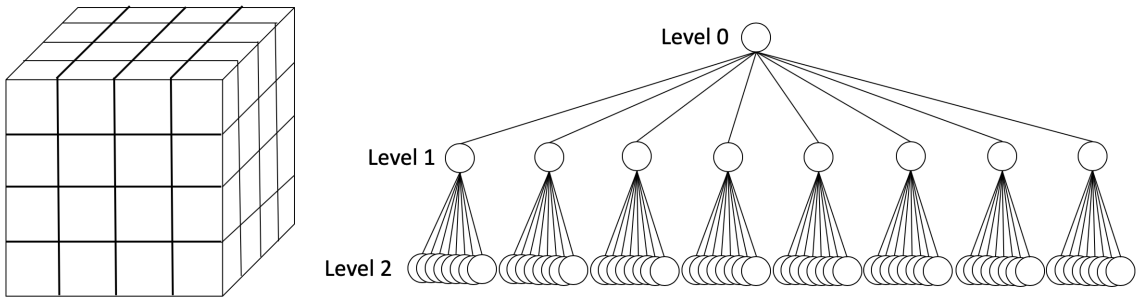


Figure 3.1: An N^3 voxel grid where $N = 4$ (left) and the corresponding octree with 2 levels (right).

Although SVOs are an efficient geometric representation, memory becomes a bottleneck for SVOs in high resolution scenes. Kämpe et al. [17] take SVOs one step further and convert them into directed acyclic graphs (DAGs). They found that Sparse Voxel DAGs represent binary voxel grids orders of magnitude more efficiently than SVOs since SVOs can have a large number of redundant subtrees. To convert SVOs into DAGs, the authors merge common subtrees, which resemble identical regions of space. This merging of subtrees allows different nodes to point to the same child. Consequently, duplicate subtrees can be removed, freeing up memory. The authors' algorithm to transform an SVO into a DAG does not interfere with the ray tracing process because the traversal path from the root node to a specific leaf node remains the same.

To reduce an SVO into a DAG, the authors propose the following method:

- Merge identical leaf nodes
- Update the child pointers in the level above to reference the unique leaves
- Repeatedly go to the next level above and merge nodes with identical childmasks and pointers

Figure 3.2 shows a diagram for this process.

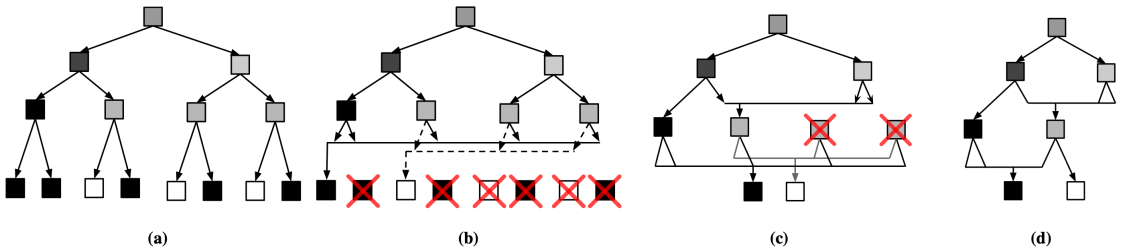


Figure 3.2: The bottom-up algorithm to reduce an SVO into a DAG. a) The original SVO. b) Merge identical leaf nodes and update the parent pointers c) Reduce non-unique nodes in the level above the leaves d) The final DAG. By Kämpe et al. [17].

Although this work cleverly constructs a memory-efficient geometric representation, Sparse Voxel DAGs do not store material information (i.e., shine constant or ambient, diffuse, and specular reflection coefficients). To obtain material information when ray tracing primary rays, the authors had to query a traditional SVO that contained the necessary material information. DAGs cannot directly store material information because there is not a one-to-one correspondence between DAG leaf nodes and voxels in the 3D scene. In other words, a DAG leaf node can represent the geometry for several voxels in a scene. In contrast, an SVO leaf node represents a single, unique voxel. Thus, SVO nodes can store a reference to material information specific to each voxel. In following sections, some follow-on works are discussed that have explored finding a separate material representation to accompany Sparse Voxel DAGs. This thesis proposes a material representation based on hashing.

Several researchers have built upon and extended different areas of Sparse Voxel DAGs. Kämpe et al. [18] used the Sparse Voxel DAG binary voxel representation to generate precomputed voxelized shadows. They also sped up DAG construction times and found improved algorithms for subtree merging. Dolonius [8] notes that the faster construction and further compressing of DAGs potentially allows the recomputation of the DAG during run time for slowly moving shadows. Villanueva et al. [24] also extend the work with Sparse Voxel DAGs by further merging subtrees deemed identical through a similarity transform. They claim that their method achieves a more efficient and lossless compression of the geometry.

3.3 Moxel DAGs

The other core paper referenced throughout this work is Brent Williams's "Moxel DAGs: Connecting Material Information to Sparse Voxel DAGs" [27]. Williams extended the work in [17] by creating a material representation for DAGs. His Moxel DAG allows equivalent renders to those with traditional SVOs while using less memory.

To connect material information to DAGs, Williams added data to each DAG node and an external table to store materials for filled voxels. The material table is ordered by Morton code such that the material information for the voxel with the smallest Morton code is the first element in the table. Williams's method stores empty node counts in each DAG node so that upon traversal of the DAG, an index into the external material table can be calculated. The empty count associated with a particular node represents the number of empty leaf voxels on the max level of the tree that come before the current voxel. In Williams's implementation, a Moxel DAG node contains a childmask and a set of pointers to the non-empty children. The mask is allocated

eight bytes to keep byte alignment despite only needing one byte. Williams uses the seven unused bytes in the mask to store empty node counts for every filled child except the rightmost child; the rightmost filled child does not need to store empty counts because its empty count is not used in any calculations. Figure 3.3 shows the structure of a Moxel DAG node. After reducing an SVO into a DAG, Williams's work traverses the DAG again to calculate and set the appropriate empty counts.

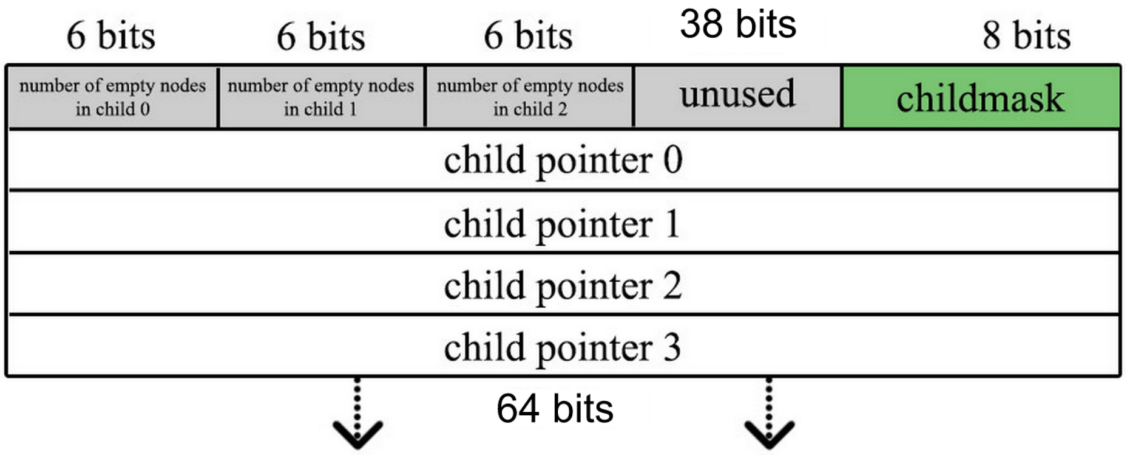


Figure 3.3: The structure of a Moxel DAG node. By Brent Williams [27].

During the ray tracing of the Moxel DAG, Williams's algorithm keeps track of two running sums: one for the voxel number, which is also the voxel's Morton code, and one for the number of empty nodes before that voxel. Both of these sums are dependent on what path is taken through the DAG. If a ray-voxel intersection occurs, an index into an external material table can be calculated by subtracting the empty count from the voxel index; Williams named this offset the moxel index. The moxel index is used to retrieve a particular voxel's material information.

$$moxelIdx = voxelIdx - emptyCount$$

Figure 3.4 shows the process for calculating the moxel index of voxel J_2. Note that there is a voxel named J_1 and another one named J_2 because there are two different paths to leaf node J in the DAG. Voxel J_1 is reached through the path A→B→D→G→J. Voxel J_2 is reached through the path A→C→F→G→J. Each filled voxel has a cell for its material information in the Moxel Table. Since voxel J_2 has a voxel number of 14 and an empty count sum of 8, its moxel index is 6 and its material information is stored in cell 6 of the Moxel Table.

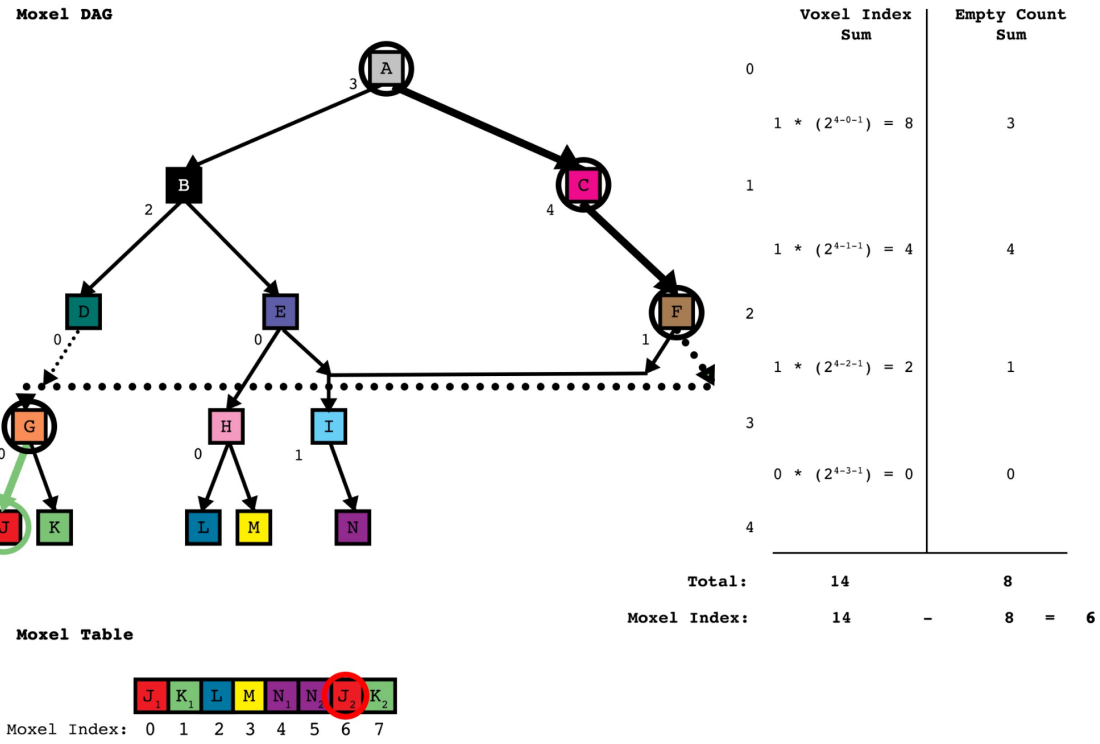


Figure 3.4: The steps to calculate voxel J_2's moxel index. By Brent Williams [27].

Although Williams's implementation successfully connects a material representation to High Resolution Sparse Voxel DAGs, it experiences some drawbacks in terms of memory, build times, and render times. For example, the storage of empty counts for each pointer becomes problematic in high resolution scenes since the number of empty nodes can reach large numbers. Williams also notes that the rendering time is 1.7

times longer with Moxel DAGs than with Sparse Voxel DAGs, and that a considerable amount of time is taken to construct the Moxel DAG and Moxel Table. The work in this thesis is motivated by Williams's suggestion in his Future Work section to build another material representation based on hashing so that empty counts no longer need to be stored.

3.4 Other Material Structures and Material Compression

A few other papers have explored material structures to accompany Sparse Voxel DAGs. In "Geometry and Attribute Compression for Voxel Scenes," Dado et al. [7] decouple voxel material attributes from the geometry. Instead of storing empty counts like in [27], the authors store offsets in the child pointers of DAG nodes. These offsets are based on a depth-first ordering of the nodes in the initial SVO. The offsets allow the calculation of an index into an external attribute array, which is compressed separately using a palette compression technique. Similar to Dado et al., Dolonius et al. [9] in "Compressing Color Data for Voxelized Surface Geometry" decouple geometry from material attributes and examine compression techniques. Instead of storing a value per pointer, the authors found a way to store a value per node; their work stores the number of voxels contained in the node's subgraph. For color compression, the authors incorporated an image compression technique by mapping a 1D array of colors onto a 2D image.

Chapter 4

IMPLEMENTATION

This chapter presents the implementation of our data structures and algorithms for ray tracing scenes with voxelized geometry. The primary goal of these structures and algorithms is to improve ray tracing efficiency in terms of memory and performance.

Specifically, this chapter shows a new material representation that can be connected to Sparse Voxel DAGs. The material representation hashes voxel indices to material properties (i.e., normal coordinates and reflectance properties). In contrast to the structures in previous works, this material representation does not require additional information to be stored in DAG nodes. To further reduce memory usage, a method is implemented to calculate a voxel's normal based on surrounding voxels. Since each voxel's normal is calculated with this method, the material representation no longer needs to store normals. Lastly, this work shows that DAGs can be used to figure out when rays intersect with closer voxels. This information allows the use of anti-aliasing on closer objects in the scene.

4.1 Starting Code Workflow

The ray tracer used in this work was constructed from scratch using C++. Triangle voxelization, SVO construction, DAG construction, and Moxel DAG construction were taken from and improved upon Williams's Moxel DAG implementation [1]. By starting where Williams left off, this work could directly extend the Moxel DAG implementation and immediately experiment with new features.

The ray tracing program's starting workflow is listed below:

1. Read and parse an OBJ file storing the scene geometric information
2. Read and parse an MTL file storing the scene material information
3. Voxelize the scene
4. Build an SVO to represent the filled voxels
5. Reduce the SVO into a Sparse Voxel DAG
6. Calculate and set the empty counts to convert DAG nodes into Moxel DAG nodes
7. Build the Moxel Table with an entry for each filled voxel in the scene
8. Ray trace the scene with 1 ray per pixel
   (a) Use the Moxel DAG for geometric queries
   (b) Use the Moxel Table for normals and material information

4.2 Connecting a New Material Representation Using Hashing

Hashing is a technique that maps keys to values. In the context of this work, a voxel index can be used as a key in a hash table to retrieve the voxel's normal and material information, which act as the value. Accessing the normal and material information through a voxel's voxel index is beneficial because no additional data needs to be stored in each DAG node or pointer. The Sparse Voxel DAG geometric representation can be used without any modifications since there is no longer a need to calculate empty counts or convert DAG nodes into Moxel DAG nodes. The hash table also eliminates the need to build the ordered Moxel Table.

4.2.1 The Voxel Index

The voxel index can be thought of as the number of leaf nodes that come before the voxel (from left to right) in a full SVO. For example, in a two-level SVO with every voxel filled, the voxel with index 0 is the leftmost leaf node and the voxel with index 63 is the rightmost leaf node. Figure 4.1 displays all voxel index labels for a full, two-level SVO.

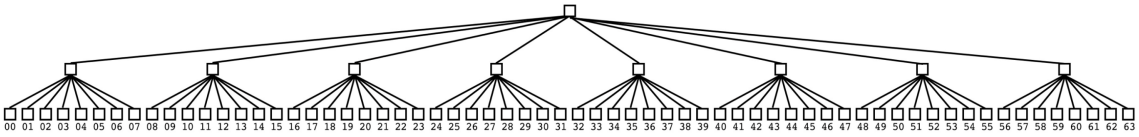


Figure 4.1: Labeled voxel indices for each leaf node in a full, two-level SVO. By Brent Williams [27].

As described in [27], a voxel's index can be calculated by keeping a running sum throughout the traversal of an SVO or a DAG. The calculation is the same for both the SVO and DAG because the traversal path to a specific leaf voxel is equivalent for both structures. Traversal starts at the root node and the running sum starts at 0. A value is added to the sum each time a child node is traversed to. This value represents the number of leaf nodes before the current node that would exist in a full SVO. In other words, it represents the number of voxels that have smaller voxel indices than any future voxel being traversed to. This value is calculated with the following equation:

$$v = c * (8^{n-p-1}),$$

where v is the value added to the running sum, c is the child index ($0 \leq c \leq 7$ for an octree), n is the max level, and p is the level of the parent node ($0 \leq p \leq n-1$). Figure 4.2 exemplifies how to calculate the voxel index for a particular voxel in an SVO.

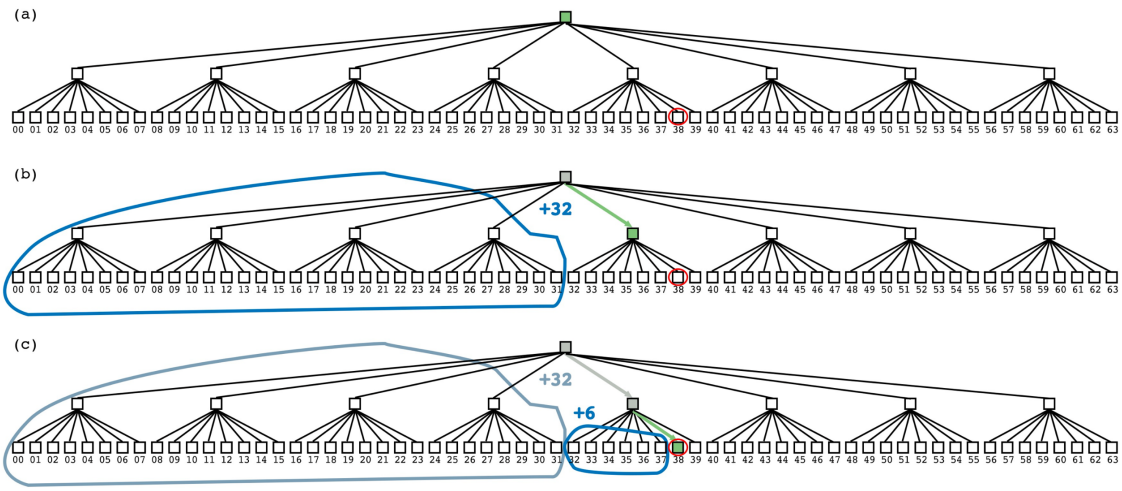


Figure 4.2: The process to calculate a voxel's index in a two-level SVO ($n = 2$). The green node represents the current node and the node circled in red represents the target node. (a) The current node starts as the root node and the running sum starts at 0. (b) There is a traversal to the index 4 child ($c = 4$ and $p = 0$) and $4 * (8^{2-0-1}) = 32$ is added to the running sum. (b) There is a traversal to the index 6 child ($c = 6$ and $p = 1$) and $6 * (8^{2-1-1}) = 6$ is added to the running sum. The final voxel index is $32 + 6 = 38$. By Brent Williams [27].

4.2.2 Hash Table Details

The `tbb::concurrent_unordered_map` container with keys of type `uint64_t` (unsigned 64-bit integer) and values of custom-type `ShadingData` is used to implement the hash table. This structure was chosen because it supports concurrent insertion. The support for concurrent insertion is necessary for multithreading during triangle voxelization since this is when entries are added into the hash table. Multithreading also occurs during the actual ray tracing since this is when material information is retrieved from the hash table. The retrieval operation for this structure is concurrency-safe because data is only being read.

The `ShadingData` class has a `glm::vec3` attribute for a voxel's normal and an `unsigned int` for a voxel's material index. Since several voxels share the same mate-

rial properties, one copy of each unique material is stored in a vector to save memory. This vector is indexed with a material index and constructed when the MTL file is parsed at the start of the ray tracing program. The material properties are represented by a `float` for the shininess constant and `glm::vec3s` for the ambient, diffuse, and specular reflectance coefficients. Figure 4.3 shows how the hash table connects voxels to material data.

The hash table material representation is constructed during triangle voxelization. Each triangle being voxelated has three vertices and a material index. When a triangle is found to intersect with a voxel, an entry is added to the hash table that maps the voxel index to the voxel material information. The voxel index can be determined by converting the x, y, and z coordinates of the hit voxel into a Morton code through Morton encoding. The material information put into the hash table includes the triangle's normal and the triangle's material index. The normal is calculated by taking the cross product of two of the triangle's edges.

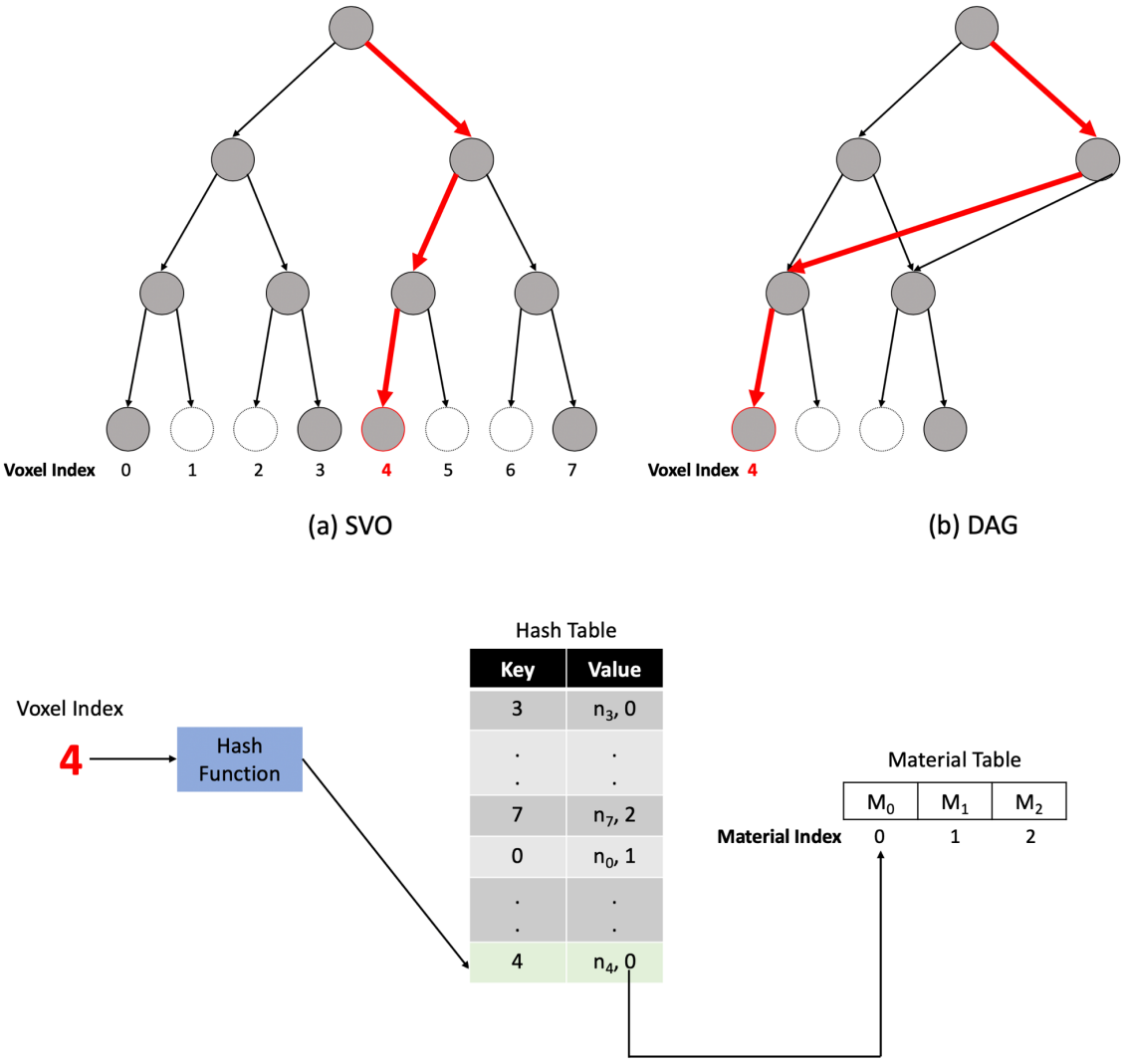


Figure 4.3: The process to get a voxel's material information. (a) The original SVO with labels for each voxel's voxel index. The colored nodes represent filled voxels and the white nodes represent empty voxels. The traversal path to the voxel with index 4 is shown in red. (b) The result of reducing the SVO into a DAG. The same traversal path for the SVO is shown in red in the DAG to the voxel with index 4. (c) To find the material information for this voxel, its voxel index, 4, is put through a hash function and matched with a key in the hash table. The value, $n_4, 0$, contains the voxel's normal, n_4, and its material index, 0. The material index is used to retrieve the material properties, M_0, from a Material Table that stores each unique material.

4.3 Calculating Normals from Surrounding Voxels

In the Moxel DAG implementation, a large portion of the total used memory is due to the material information stored in the Moxel Table. One way to reduce this memory is to decrease the amount of information stored. In the following subsections, algorithms are presented that calculate voxel normals based on the occupancy of neighboring voxels. As a result, the hash table material representation no longer needs to store normals for the filled voxels.

4.3.1 Immediate Neighbor Gradient Approach

First, a simple approach was examined to determine the plausibility of using the Sparse Voxel DAG to calculate voxel normals. Since the DAG encodes a binary voxel grid, it stores information on which voxels contain geometry; each voxel is represented by 1 bit (1 if the voxel contains geometry and 0 if the voxel is empty). When calculating a surface's normal, it is particularly helpful to determine which regions (i.e., voxels) around the surface are part of the surface.

This method uses information from six immediate voxel neigbors: left, right, bottom, top, back, and front. Figure 4.4 shows a diagram of a voxel and these immediate neighbors.

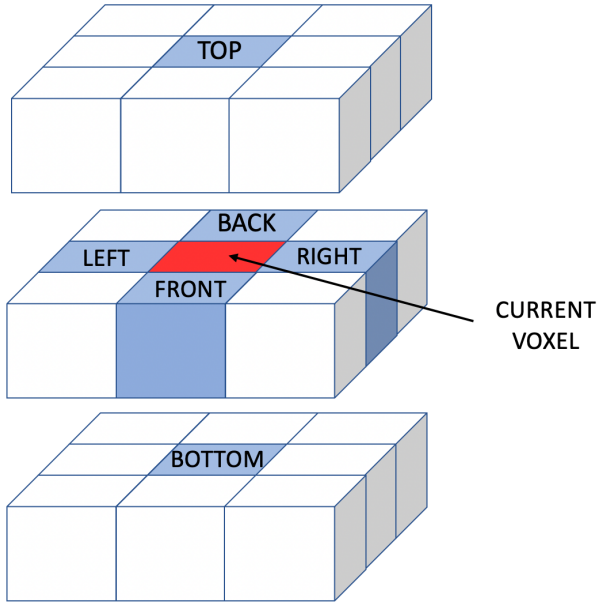


Figure 4.4: The left, right, bottom, top, back, and front neighbors of a particular voxel. The current voxel is colored red and the neighbors are colored blue.

Gradients in the x, y, and z directions can be calculated to approximate the normal by using the binary voxel information stored in the DAG; the left and right voxel neighbors can determine the normal's x-value, the bottom and top voxel neighbors can determine the normal's y-value, and the front and back voxel neighbors can determine the normal's z-value. The idea behind this method is that a surface's normal will have a higher magnitude in the direction where there is no geometry. For example, the normal of an XZ plane would be strong in the y-direction. The following formulas are used to calculate a voxel's normal:

$$normal_x = voxel_{left} - voxel_{right}$$

$$normal_y = voxel_{bottom} - voxel_{top}$$

$$normal_z = voxel_{back} - voxel_{front}$$

$$normal = normalize(< normal_x, normal_y, normal_z >)$$

The value of a voxel in the formulas is the value of its bit in the Sparse Voxel DAG. Thus, if $voxel_{left}$ is filled and $voxel_{right}$ is empty, then $voxel_{left} = 1$, $voxel_{right} = 0$, and $normal_x = voxel_{left} - voxel_{right} = 1 - 0 = 1$. Morton coding enables a simple way to determine the desired neighbors and access each neighbor's bit of information in the DAG. As mentioned earlier, the voxel index of the current voxel can be calculated by keeping a running sum throughout traversal of the DAG. The voxel index is also the voxel's Morton code. A voxel's x, y, and z coordinates in a 3D voxel grid can be extracted by decoding the Morton code. With the x, y, and z coordinates of the current voxel, the x, y, and z coordinates of the desired neighbors can be determined. For example, the right voxel neighbor would have the coordinates x+1, y, and z. The neighbor coordinates can be converted back into a Morton code, which represents the neighbor's voxel index. Lastly, the DAG is traversed to the neighbor voxel index to determine if the target voxel is filled or not. The process for finding a current voxel's neighbor in the DAG is summarized below:

- Convert the current voxel's index (Morton code) into x, y, and z values
- Determine the x, y, and z coordinates of the desired neighbor relative to the current voxel's x, y, and z coordinates. For example, the top neighbor's coordinates are x, y+1, z
- Convert the neighbor voxel's x, y, and z coordinates into a voxel index (Morton code)
- Traverse the DAG to the neighbor's voxel index to determine if the neighbor has geometry or is empty

Figure 4.5 shows an image rendered with the immediate neighbor gradient technique. The lighting and shading of the bunny indicates that the calculated normals for each voxel are generally in the correct direction. However, the shading is not smooth. One limitation of this approach is that normals can only have discrete values since each dimension before normalization is either -1, 0, or 1. Another limitation is that diagonal neighbors and further neighbors are not taken into account. Although the image does not look too appealing, it serves as a proof of concept for calculating normals from voxel data. Since this idea showed potential, other research efforts were explored. In the next subsection, this work presents a method that approximates surface normals more accurately.

Figure 4.5: A render of a bunny that has its normals calculated with the immediate neighbor gradient approach. This output image shows a lot of noise and shading discontinuities, but it demonstrates the potential to approximate voxel normals from surrounding voxels.

4.3.2 Tangent Plane Approach

The next method implemented was inspired by [21] and [14]. Both of these works construct tangent planes that approximate geometric surfaces. The building of tangent planes is applicable to this work because the tangent plane's normal can be used as an approximation for the surface normal at a given voxel.

This work closely follows Ernerfeldt's algorithm [14], which fits a plane to a list of noisy 3D points. Ernerfeldt's method is summarized below:

1. Calculate the centroid of a list of points
2. Calculate the covariance matrix of the points relative to the centroid
3. Perform a linear regression on each axis (x, y, and z)
4. Weight the results of the regression by the square of the determinant

To use this algorithm, a list of points had to be constructed to approximate the surface around the voxel of interest. This list of points was created by including the x, y, and z coordinates for the filled neighbor voxels within a certain-sized voxel radius from the current voxel. The voxels within a voxel radius can be thought of as voxels within a $(2 * radius + 1)$x$(2 * radius + 1)$x$(2 * radius + 1)$ bounding cube centered at the current voxel. For example, a voxel radius of 1 (3x3x3 bounding cube) would include all the voxels shown in Figure 4.4. For each eligible neighbor, the Sparse Voxel DAG is traversed to determine if the neighbor voxel is filled or not. If the neighbor voxel is filled, then its x, y, and z coordinates are added to the list. Empty voxels are ignored since the algorithm is only interested in the voxels that approximate the surface. Neighbor voxels that do not fit inside the boundaries of the 3D voxel grid are also ignored.

The left image in Figure 4.6 shows the initial render with this method to calculate normals. The shading looks smoother than the shading from the first method, but some regions of the bunny look incorrect. Muniz et al. [21] point out that tangent planes can have another normal by inverting all signs of the original normal vector. To see if the render had some normals in the wrong direction, the image on the right in Figure 4.6 was created by flipping all the normals. The two images indicate that the incorrectly shaded bunny regions in one image can be fixed by flipping the normals for those regions.

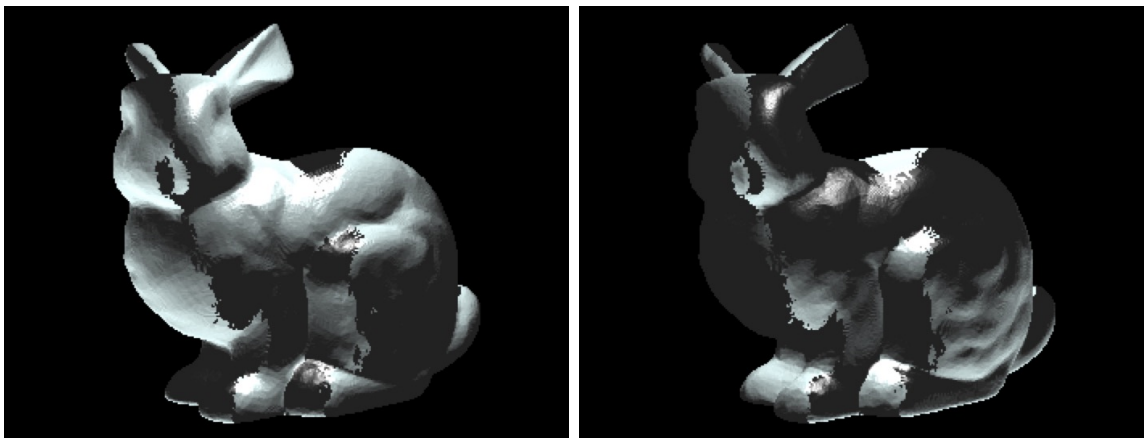
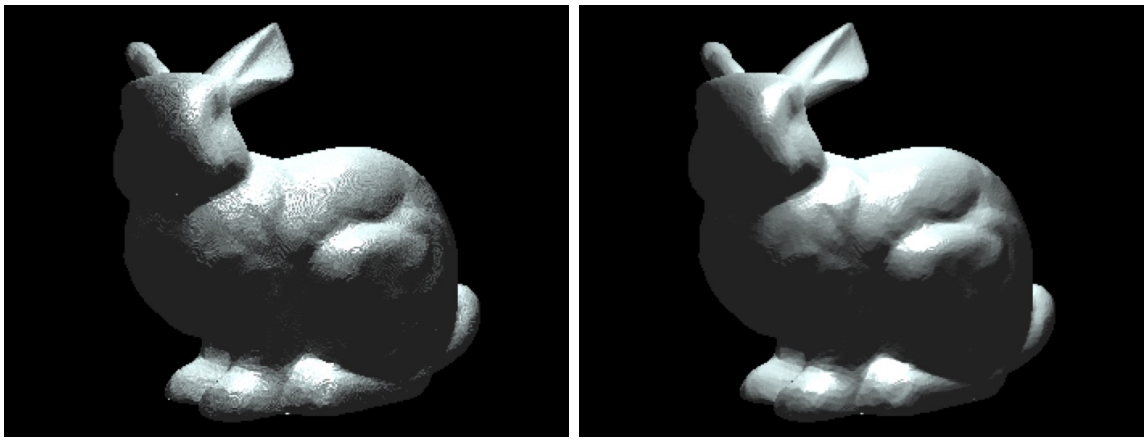


Figure 4.6: Using the tangent plane normals (left) and inverting the tangent plane normals (right).

Muniz et al. [21] choose the appropriate normal by casting two additional rays in the direction of both normals. They pick the direction that goes through less solid material. Instead of casting two more rays, this work determines which voxel cube face that the ray intersects with and picks the direction that makes the normal go toward this face. For example, if a ray intersects with the right face of a voxel, the normal vector with the positive x-value is chosen. Figure 4.7 shows the image with corrected normals.

Figure 4.7: Bunny image with the appropriate normal directions chosen.

As mentioned earlier, the number of neighbor voxels to consider when calculating the tangent plane is configurable. Figure 4.8 shows renders with voxel radii of 1 and 2.



(a) voxel radius = 1 (b) voxel radius = 2

Figure 4.8: Calculating normals using neighbors within one voxel of the current voxel (left) vs. calculating normals using neighbors within two voxels of the current voxel (right).

4.4 Super Sampling by Proximity

Since each pixel is a square and its color is determined by a single ray, the edges on ray traced objects can appear noisy or jagged (See Figure 4.9).

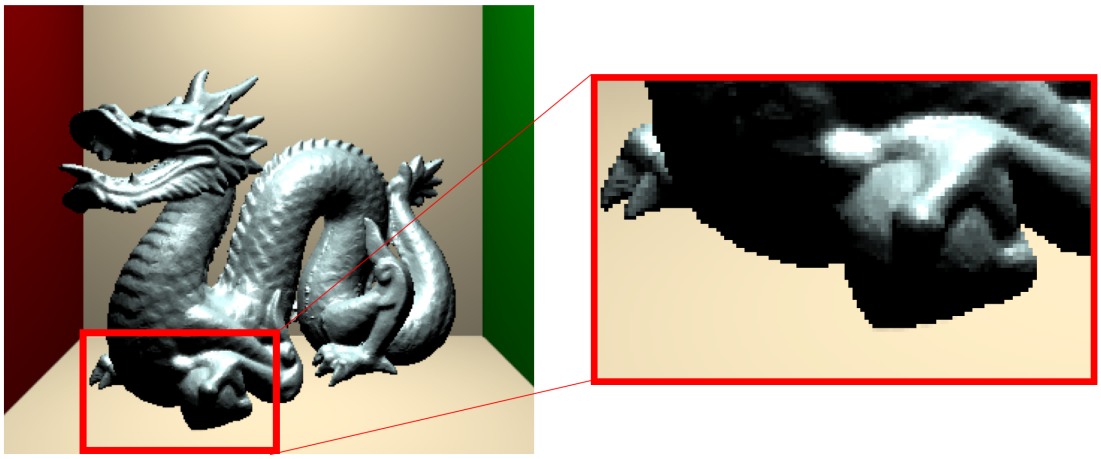


Figure 4.9: The dragon image without super sampling and a closeup of the jagged edges.

One way to reduce aliasing is through super sampling. For the super sampling method, an individual pixel is divided into an n x n grid. A ray is shot through each cell in the pixel and the colors from the n^2 rays are averaged for each pixel. With super sampling, edge colors blend with background colors to produce smoother transitions. Figure 4.10 shows the dragon image created by using a 4x4 grid of super samples for each pixel.

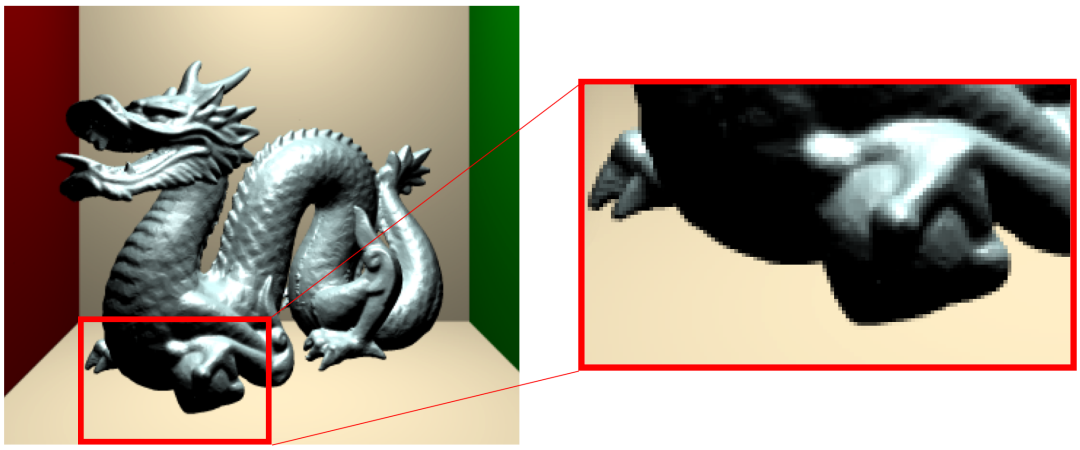


Figure 4.10: The dragon image with super sampling and a closeup of the smoother and blended edges.

Although super sampling produces subjectively better images, it increases the computational cost since more rays have to be processed. Super sampling with a 4x4 grid of super samples for each pixel is predicted to be 16 times the original work. This work proposes to only perform super sampling on certain parts of the scene to reduce some computation. It would be ideal if input OBJ files specified which areas to give more attention and detail. Since there is no such specification, a general approach was implemented to perform super sampling on pixels that represent closer locations in the scene. Once an intersection is detected, Morton decoding is used on the voxel index to figure out the x, y, and z coordinates of the hit voxel. Since each coordinate ranges from 0 to $n-1$, where n is the number of voxels that span one side of the 3D voxel grid, it is possible to determine which voxels' z coordinates are close to the virtual camera. A voxel's z coordinate is in the top 50% of closest z coordinates for the entire voxel grid if the z coordinate is greater than $(n-1)/2$. If the current voxel's z coordinate is greater than $(n-1)/2$, the pixel is super sampled and the colors from all the rays are averaged. If the z coordinate is less than or equal to $(n-1)/2$, then the color from the single ray sample is used.

Figure 4.11 shows a scene with the closest voxels colored in red and the same scene with super sampling on the red region.

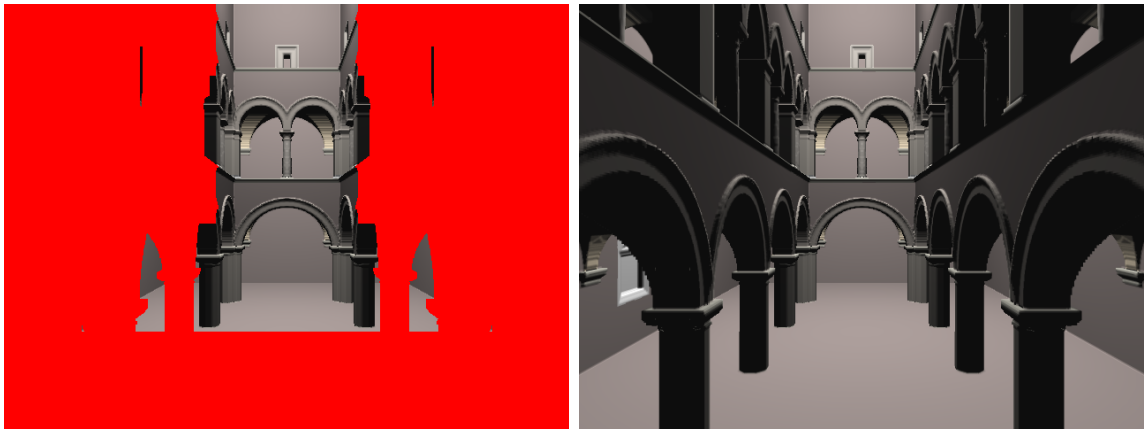


Figure 4.11: Sponza scene showing the locations that get super sampled. The red region represents filled voxels in the front half of the 3D voxel grid (left). The right image shows the same Sponza scene with super sampling performed on the red region.

4.5 Final Code Workflow

This work has examined new ways to represent and connect the geometry and materials used for ray tracing voxelized scenes. First, this work created a hash table material structure to accompany Sparse Voxel DAGs during ray tracing. The new material representation eliminates the need to construct a Moxel Table and calculate empty counts. This work also implemented a method to calculate voxel normals using the Sparse Voxel DAG to optimize memory usage. Lastly, this work demonstrated the ability to super sample certain regions of a scene, allowing anti-aliasing on closer objects.

The updated and final workflow of this work's ray tracing program is listed below:

1. Read and parse an OBJ file storing the scene geometric information

2. Read and parse an MTL file storing the scene material information

3. Voxelize the scene and construct the hash table

4. Build an SVO to represent the filled voxels

5. Reduce the SVO into a Sparse Voxel DAG

6. Ray trace the scene with multiple rays for closer voxels

    (a) Use the Sparse Voxel DAG for geometric queries and to calculate normals

    (b) Use the hash table for material information

Chapter 5

RESULTS AND VALIDATION

This thesis has presented a new material representation for Sparse Voxel DAGs, a method to approximate normals, and a technique to super sample certain regions of a scene. This section analyzes how each part of this work affects memory and performance and provides comparisons with Moxel DAGs. This section also includes the results of a user survey regarding the visual quality of this work's output images.

5.1 Test Environment

The ray tracing program was run on a 2020 MacBook Air with 16 GB of RAM and a 1.1 GHz Quad-Core Intel Core i5 processor. Like the Moxel DAG implementation, the code for this thesis is written in C++ and uses Threading Building Blocks for multithreading.

5.2 Comparisons

This work is compared with the Moxel DAG implementation to contextualize the effectiveness (or ineffectiveness) of the changes presented in this thesis. The Moxel DAG code [1] was ran on the same laptop to control for the testing environment. A few modifications were made to Williams's code so that the implementations could be compared more accurately. For example, the ray generation functionality was altered so that both ray tracers would shoot rays to the same locations in each pixel. The Moxel DAG ray tracer was also changed to produce images that matched this work's

output image sizes (640 x 480) so that both ray tracers would trace the same number of rays. Lastly, the Moxel DAG ray tracer was modified to use a single light source.

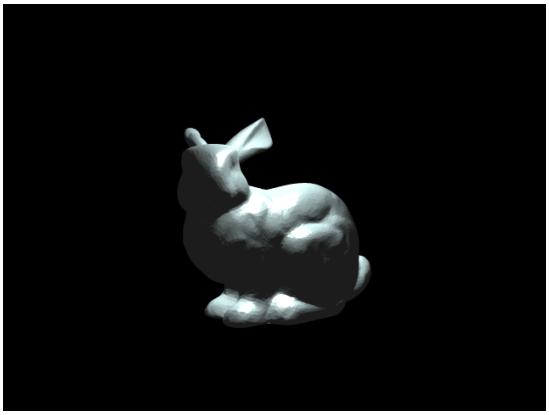
5.3 Benchmark Scenes

The results were collected by using five different benchmark scenes: Bunny, Buddha, Dragon, Sponza, and Conference Room. Figure 5.1 shows the renders of these benchmark scenes using the hash table material representation, normals calculated for each voxel, super sampling for all pixels, and one point light.

The bunny, Buddha, and dragon scenes contain remeshed Stanford models. These test scenes were taken directly from the Moxel DAG code base [1]. Note that the Toy Store scene from the Moxel DAG implementation was not used because it was not included in the GitHub repository. To test the ray tracers on bigger scenes with more geometry and materials, the Dabrovic Sponza and Conference Room meshes were acquired from the McGuire Computer Graphics Archive [20]. Blender was used to import both of these meshes, edit out some geometry, and orient the models. Blender was also used to subdivide the Sponza model to increase its triangle count. Table 5.1 provides specific information about each benchmark scene.

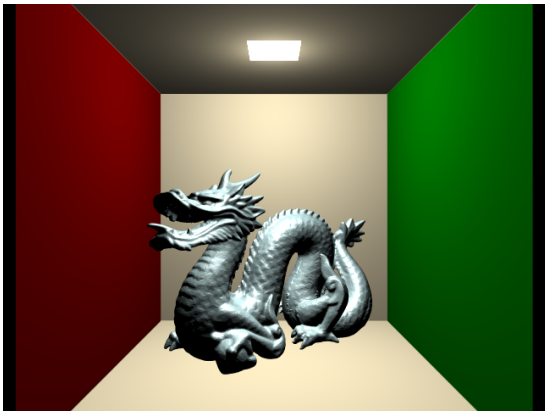
Table 5.1: The number of triangles and materials for each benchmark scene.

Scene	Number of Triangles	Number of Materials
Bunny	29,822	1
Buddha	100,014	1
Dragon	100,012	5
Sponza	328,131	19
Conference	322,457	32



(a) Bunny

(b) Buddha



(c) Dragon

(d) Sponza

(e) Conference Room

Figure 5.1: The benchmark scenes used for testing the ray tracer.

5.4 Analysis

In the following sections, the main three parts of this thesis are analyzed with a focus on memory and running times.

5.4.1 Part 1: Hash Table Material Representation

This part examines how the hash table material representation compares to the Moxel DAG implementation in terms of memory, build times, and render times. Both ray tracers were run on the five benchmark scenes at three different voxel resolutions (256^3, 512^3, and 1024^3). Each timing statistic is an average of three program runs.

5.4.1.1 Memory

The memory results are shown in Table 5.2. The left table compares the Sparse Voxel DAG in this work to the Moxel DAG in Williams's implementation. The results show that the Sparse Voxel DAG memory is only a fraction of the Moxel DAG memory in all scenarios. An advantage of the hash table material representation is that no extra data needs to be stored in each node to retrieve the correct material information. In contrast, Moxel DAGs store empty counts for each pointer so that an offset can be calculated into the external Moxel Table. The difference in memory between Sparse Voxel DAGs and Moxel DAGs is solely due to the empty counts.

The right part of Table 5.2 shows the memory comparisons between the hash table and Moxel Table. In all scenarios, the hash table uses more memory than the Moxel Table. The increased memory is due to the fact that hash tables have to store keys along with the values. The hash table has keys of type `uint64_t` (unsigned 64-bit integer) because each key is a voxel index. Since both tables have an entry for every

voxel that has geometry, the hash table increasingly uses more memory than the Moxel Table as the number of filled voxels increases. It is important to note that the material representations use significantly more memory than the DAGs. In Table 5.2, the material representations are measured in MB whereas the DAGs are measured in KB. The memory saved by using Sparse Voxel DAGs without empty counts is not nearly enough to counteract the increased memory resulting from the hash table's storage of keys for filled voxels. Overall, this work's method uses roughly 1.49 times the memory of the Moxel DAG implementation.

Table 5.2: Comparing the memory requirements between this work's implementation (Sparse Voxel DAG, Hash Table) and Williams's implementation (Moxel DAG, Moxel Table) on different scenes at different resolutions.

		Memory (KB)		
Scene		256^3	512^3	1024^3
Bunny	Sparse Voxel DAG	144	497	1712
	Moxel DAG	178	639	2262
Buddha	Sparse Voxel DAG	100	377	1378
	Moxel DAG	127	487	1806
Dragon	Sparse Voxel DAG	156	566	1987
	Moxel DAG	196	727	2610
Sponza	Sparse Voxel DAG	144	368	828
	Moxel DAG	221	598	1365
Conference	Sparse Voxel DAG	76	212	634
	Moxel DAG	104	303	905

(a) Sparse Voxel DAG vs. Moxel DAG Memory

		Memory (MB)		
Scene		256^3	512^3	1024^3
Bunny	Hash Table	12	47	188
	Moxel Table	8	31	125
Buddha	Hash Table	11	44	177
	Moxel Table	7	30	118
Dragon	Hash Table	10	42	169
	Moxel Table	7	28	113
Sponza	Hash Table	14	56	226
	Moxel Table	10	38	151
Conference	Hash Table	5	20	87
	Moxel Table	3	14	58

(b) Hash Table vs. Moxel Table Memory

5.4.1.2 Build Times

The build times for this work's data structures are shown in Table 5.3. The timings for voxelization, hash table construction, and DAG construction are included. Note

that voxelization and hash table construction are timed together since the hash table material representation is constructed during voxelization.

The build times for Williams's data structures are shown in Table 5.4. This table includes the time for voxelization, Moxel DAG construction, and Moxel Table construction. Williams does not build a hash table material representation in his voxelization step, but he does create another map structure that this work's voxelization algorithm does not; when a triangle intersects with a voxel, Williams's method maps the voxel index to the triangle's index in a vector of all triangles. Williams's work later uses this map to fill the Moxel Table with material information for each voxel that contains geometry.

Table 5.3: My build times

		My Build Times (ms)		
Scene		256^3	512^3	1024^3
Bunny	Voxelization + Hash Table Construction	248	957	4028
	DAG Construction	61	842	9439
Buddha	Voxelization + Hash Table Construction	290	979	3929
	DAG Construction	41	678	9497
Dragon	Voxelization + Hash Table Construction	315	1007	4236
	DAG Construction	59	900	11872
Sponza	Voxelization + Hash Table Construction	651	1618	5313
	DAG Construction	45	440	4238
Conference	Voxelization + Hash Table Construction	284	653	2191
	DAG Construction	15	156	1703

Table 5.4: Moxel build times

		Moxel Build Times (ms)		
Scene		256^3	512^3	1024^3
Bunny	Voxelization	440	2306	16032
	Moxel DAG Construction	303	3282	37443
	Moxel Table Construction	1368	11590	101376
Buddha	Voxelization	608	3154	20371
	Moxel DAG Construction	331	3391	39008
	Moxel Table Construction	1380	11708	100898
Dragon	Voxelization	567	2745	18762
	Moxel DAG Construction	282	3062	37292
	Moxel Table Construction	1183	10506	92286
Sponza	Voxelization	956	3946	26689
	Moxel DAG Construction	289	2839	33785
	Moxel Table Construction	1155	9511	90293
Conference	Voxelization	441	1070	4963
	Moxel DAG Construction	211	1987	22444
	Moxel Table Construction	839	7536	74245

This work's build times for each benchmark scene are significantly faster than the Moxel implementation build times. It was particularly interesting that this work's voxelization times were roughly two to four times faster than Williams's voxelization times. Since both algorithms perform the same intersection tests on each triangle and construct maps with voxel indices as keys, the times were expected to be similar. After examining both algorithms more closely, a few differences were found that likely contributed to the time difference. First, during triangle voxelization, the Moxel DAG implementation constructs a bigger bounding cube around the triangle than this work does. Consequently, the Moxel DAG implementation performs several more voxel-triangle intersection tests. As another difference, the Moxel DAG implementation

uses a custom `Vec3` class to represent 3D vectors whereas this work uses the `GLM` library.

The results also show that Moxel DAG and Moxel Table construction take a large percentage of the total build time. As expected, the Moxel DAG construction takes longer than the Sparse Voxel DAG construction. The Moxel DAG build time consists of the time to build the Sparse Voxel DAG and the time to traverse the Sparse Voxel DAG to fill it with empty counts. The Moxel Table construction time is so large because it loops through every single voxel, tests if the voxel contains geometry, and writes the material information into the table if the voxel is full.

Overall, this work's method to connect material information to the Sparse Voxel DAG with a hash table produces a big speedup in build times. A big advantage of this method is that the material representation can be constructed quickly during voxelization. The timings even show that the voxelization, hash table construction, and Sparse Voxel DAG construction times together are only a percentage of the Moxel Table construction times.

5.4.1.3 Render Times

The rendering times in Table 5.5 represent how long it took to ray trace and produce a 640x480 output image with five samples per pixel.

Table 5.5: Sparse Voxel DAG vs. Moxel DAG render times

		Render Times (ms)		
Scene		256^3	512^3	1024^3
Bunny	Sparse Voxel DAG	4104	4769	6446
	Moxel DAG	6856	8277	10117
Buddha	Sparse Voxel DAG	3850	4580	7086
	Moxel DAG	6555	8058	9308
Dragon	Sparse Voxel DAG	3915	4600	7054
	Moxel DAG	5651	7051	8744
Sponza	Sparse Voxel DAG	11589	12814	18826
	Moxel DAG	23998	28934	35867
Conference	Sparse Voxel DAG	4426	5214	7888
	Moxel DAG	8256	10165	13925

Similar to Williams's finding, the Moxel DAG takes on average about 1.7 times longer to render images than the Sparse Voxel DAG. The difference in times can be attributed to the different methods for retrieving material information. For this work's implementation, a running sum is kept during DAG traversal to determine the voxel index. This voxel index is used to retrieve materials from a hash table. For the Moxel DAG implementation, two running sums are kept during DAG traversal for the voxel index and the empty counts. These values are used to calculate an offset to index into the Moxel Table.

5.4.1.4 Part 1 (Hash Table) Summary

The use of a hash table allows the use of the Sparse Voxel DAG without any changes, enables elements to be inserted in any order, and reduces the number of calculations

during ray tracing. The results show that the hash table material representation performs better than the Moxel DAG in terms of build times and render times for all benchmark scenes and all voxel resolutions.

On the other hand, the hash table material representation uses more memory than the Moxel DAG implementation. Initially, it was predicted that the hash table method would save memory overall by removing the need to store empty counts in the DAG. Although the Sparse Voxel DAG did use less memory than the Moxel DAG, the hash table took up much more memory than expected. The material data structures (hash table and Moxel Table) use significantly more memory than the geometric data structures (Sparse Voxel DAG and Moxel DAG). Since this work's method increased the memory of the material data structure, the reduced memory from the geometric structure proved insignificant. The large memory requirements of the hash table inspired Part 2 of this work, in which the hash table's memory footprint is reduced.

5.4.2 Part 2: Calculating Own Normals

This section analyzes the effects of calculating normals from surrounding voxels instead of saving normals in the hash table. For these results, a voxel radius of two was used when calculating a voxel's normal. Build times are not included in the analysis since the hash table construction only required a minor change and the DAG construction required no changes.

5.4.2.1 Memory

The memory of the hash table material representation without normals is shown in Table 5.6. In comparison to the hash table with normals and the Moxel Table (See Table 5.2), the hash table without normals uses significantly less memory. It uses

about 50% of the hash table with normals memory and about 75% of the Moxel Table memory. Storing normals in the material data structure is memory expensive because it requires three `float` values for every voxel with geometry.

Table 5.6: The memory for the hash table without normals. This updated hash table maps a voxel index to a material index.

		Memory (MB)		
Scene		256^3	512^3	1024^3
Bunny	Hash Table (w/o normals)	6	23	94
Buddha	Hash Table (w/o normals)	5	22	89
Dragon	Hash Table (w/o normals)	5	21	84
Sponza	Hash Table (w/o normals)	7	28	113
Conference	Hash Table (w/o normals)	2	10	43

5.4.2.2 Render Times

Although calculating normals for each voxel saves a bunch of memory, it comes at the cost of rendering time. Previously, voxel normals were quickly retrieved from the hash table. The process of calculating a voxel's normal requires traversing the DAG several times to determine which neighbor voxels are full for approximating a tangent plane. The updated render times with this process are included in Table 5.7. The timings reveal that the normal calculations increase the render time in all scenarios when compared to the Part 1 implementation render times (See Table 5.5). When compared to the Moxel DAG render times (Table 5.5), the updated render times are somewhat similar, but slightly larger, for most of the benchmark scenes. One exception is the Sponza benchmark as the updated render times are smaller than the Moxel DAG render times for all three voxel resolutions. One likely explanation is that more ray-voxel intersections occur in the Sponza scene since most of the visible space

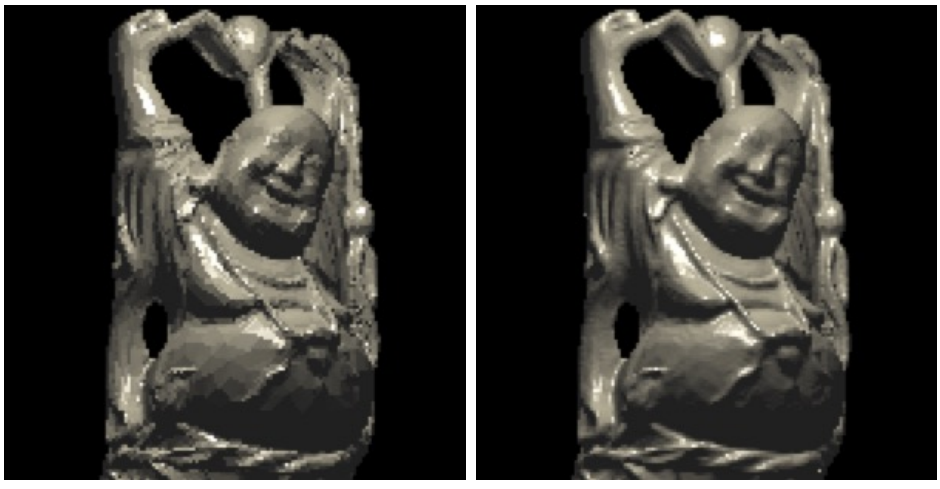
in the image is filled by geometry. As a result, the Moxel DAG implementation would spend more time calculating the running sum for empty counts during ray tracing.

Table 5.7: The render times for the ray tracer when calculating normals for each filled voxel instead of retrieving saved normals in the hash table.

		Render Times (ms)		
Scene		256^3	512^3	1024^3
Bunny	Calculated Normals	7008	8996	11180
Buddha	Calculated Normals	6740	8219	10453
Dragon	Calculated Normals	6652	8215	10117
Sponza	Calculated Normals	18300	20372	23080
Conference	Calculated Normals	8525	9366	11132

5.4.2.3 Visual Quality

Ray tracing with calculated normals from surrounding voxels altered the output images in terms of visual quality. Figure 5.2 shows what one of the benchmark scenes looks like before and after this modification.



(a) Before (b) After

Figure 5.2: Using triangle face normals saved in the hash table (left) vs. calculating normals from surrounding voxels (right).

The left image uses the normals saved in the hash table. The saved normal for a particular voxel is the face normal of the triangle that intersected with that particular voxel. Thus, voxels that approximate the same triangle get the same normal. Several areas in the left image, like the Buddha's belly, are segmented and shaded flatly since the voxels that make up each triangle use the same normal. The right image has a smoother look because normals are being calculated for each specific voxel, allowing a more accurate approximation of the surface.

5.4.2.4 Part 2 (Calculating Own Normals) Summary

The Part 2 results indicate that calculating normals from surrounding voxels instead of saving the triangle face normals in the hash table saves a significant amount of memory. However, calculating normals is extra work that causes the render time to become slightly slower than the render times with Moxel DAGs for most benchmark scenes. It was also shown that calculating normals for each voxel alters the shading on the output images. Part 3 of this work also affects the visual quality of the output images.

5.4.3 Part 3: Super Sampling

Figure 4.10 exemplified how super sampling a scene could produce smoother and blended edges. Super sampling times are presented here to quantify the cost of super sampling and to compare full super sampling to this work's method of super sampling by proximity.

5.4.3.1 Render Times

Table 5.8 shows the super sampling timings for this work's program executions. The scenes were timed with 1 sample per pixel (spp) as a baseline. Timings with 16 spp for every pixel and 16 spp for pixels representing geometry in the closest 50% of voxels were also collected.

Table 5.8: The render times with different levels of super sampling (1 spp, 16 spp, and 16 spp for the closest 50% of voxels).

		Render Times (ms)		
Scene		256^3	512^3	1024^3
Bunny	1 samples per pixel	1798	2056	2574
	16 samples per pixel	30587	35206	41169
	16 samples per pixel (closest 50%)	14569	17481	21275
Buddha	1 samples per pixel	1766	2013	2492
	16 samples per pixel	29720	35002	39399
	16 samples per pixel (closest 50%)	13189	15937	19203
Dragon	1 samples per pixel	1642	1976	2477
	16 samples per pixel	26831	34787	40861
	16 samples per pixel (closest 50%)	15667	17388	20802
Sponza	1 samples per pixel	3985	4634	5526
	16 samples per pixel	72406	80157	94320
	16 samples per pixel (closest 50%)	61177	65439	75672
Conference	1 samples per pixel	1906	2286	2793
	16 samples per pixel	34974	39620	43847
	16 samples per pixel (closest 50%)	17506	21124	23397

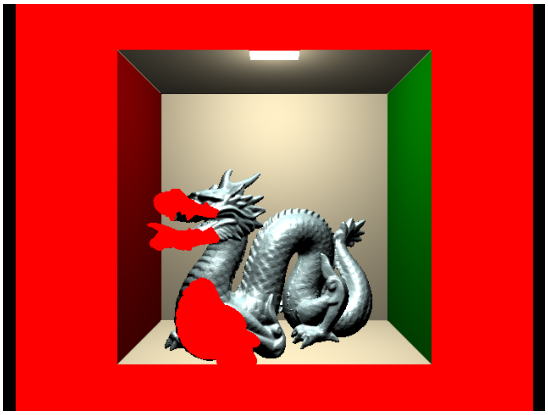
The timing results reveal that super sampling is a computationally expensive process. They also validate the expectation of the render time to increase by a factor of the number of ray samples per pixel. For example, super sampling with 16 samples per pixel roughly increases the render time by a factor of 16. This finding makes sense intuitively since ray tracing 16 rays per pixel versus one ray per pixel is 16 times the work.

The results also show that only super sampling pixels with close ray-voxel intersections reduces the time compared to super sampling all pixels. The super sampling by proximity method takes about half the time for all benchmark scenes except the Sponza scene.

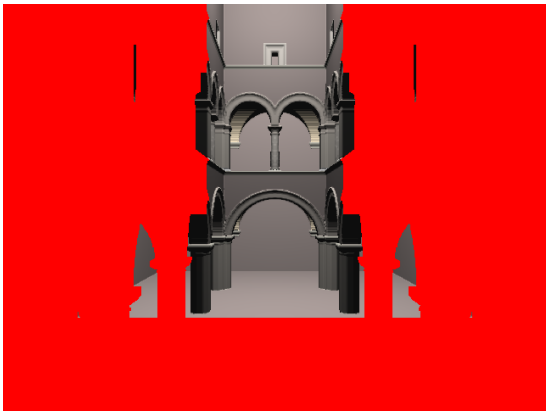
The Sponza scene differs from the other benchmark scenes in that a large percentage of the rendered image is closer to the virtual camera. Figure 5.3 displays the regions in red that are super sampled for each of the benchmark scenes. The Sponza image is mostly red, indicating that most of the image is nearby geometry. Note that the bunny and Buddha images contain red regions around the models. These red outlines occur because the bunny and Buddha models are encapsulated in a box. Although super sampling the closest geometry does well for the Sponza and Conference scenes, this method has limitations. For example, none of the Buddha model gets super sampled since it is spatially located in the back half of the scene.

(a) Bunny

(b) Buddha



(c) Dragon



(d) Sponza

(e) Conference Room

Figure 5.3: Regions that get super sampled for each benchmark scene with the super sampling by proximity method are colored in red.

5.4.3.2 Qualitative Comparison

Some of the results from super sampling by proximity can be seen in Figure 5.4. The left column contains ray traced images using 16 samples per pixel on nearby geometry. The right column contains ray traced images using 16 samples per pixel for every pixel. The images are placed next to each other to highlight any differences.

After an initial glance, the two techniques seem to produce nearly identical images. It is difficult to find differences on the full-sized images by eyeballing them. This difficulty reveals that super sampling by proximity can produce comparable renders. Although not examined in this thesis, one method to identify differences is to compare the pixel values between the two images. Zooming into specific regions in the back half of scenes reveals that the fully super sampled approach reduces noise and produces smoother images; the head and neck outlines of the dragon appear less jagged, the half-circle ridges in the back seem smoother in the Sponza image, and the back chairs seem to be blended more smoothly in the Conference scene.

5.4.3.3 Part 3 (Super Sampling) Summary

Overall, the results show that ray tracing with more rays significantly increases rendering time. If speed is a concern, super sampling by proximity is a viable method. It works best when the most important objects are located closer in the scene, but the algorithm can easily be tuned to super sample localized regions other than the front half. For example, the algorithm can be modified to super sample the right side of images or the top-right corner. Although super sampling every pixel produces subjectively higher quality images, selectively super sampling by proximity reduces render time and produces comparable output images.

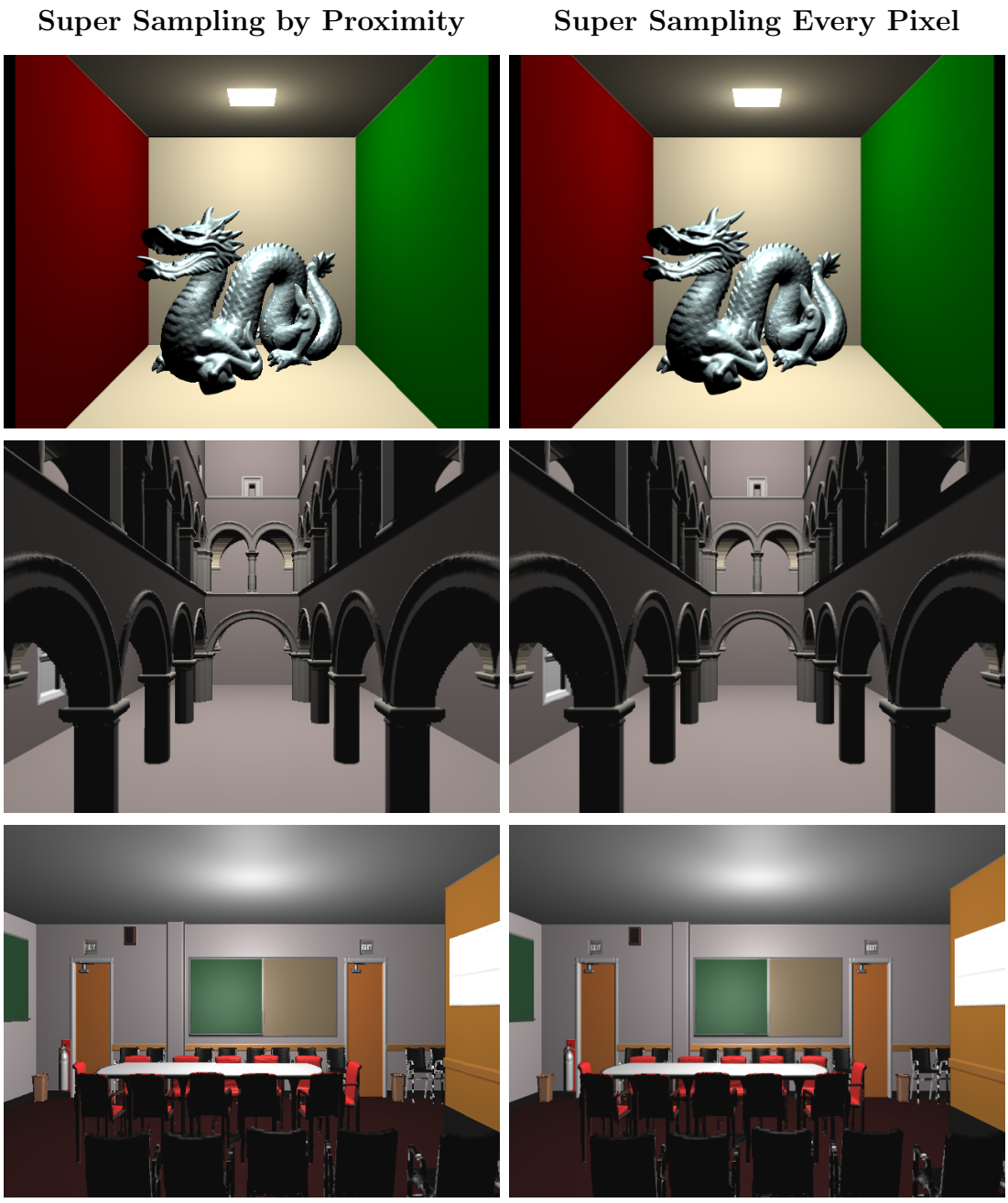


Figure 5.4: Images created by super sampling pixels that represent geometry in the front half of the scene (left) vs. images created by super sampling all pixels (right).

5.5 Visual Quality Survey

Since several of this work's algorithms affect the quality of the output images, a survey was crafted and dispensed to learn about how people would rank the techniques. A Google Forms survey was sent to people in the Cal Poly Graphics Group, some students in a Cal Poly introductory computer science course (CPE 101), and a few friends. Respondents were allowed and encouraged to send out the survey link to any of their friends. The survey had a total sample size of $n = 31$.

5.5.1 Survey Content

The survey consists of five main parts (one for each benchmark scene). In each part, four images of a cropped and zoomed in region of each benchmark scene were displayed. Each image was rendered using one of four configurations: using the triangle face normals (baseline), calculating normals with a voxel radius of one (neighbor1), calculating normals with a voxel radius of two (neighbor2), and super sampling every pixel with 16 rays per pixel (super sampling). The baseline configuration is the one used in the Moxel DAG implementation [27]. The neighbor1 and neighbor2 configurations are the ones used in this thesis to approximate the tangent plane of a surface from surrounding voxels. The super sampling configuration calculates normals with a voxel radius of two and sends 16 rays per pixel to determine each pixel's color.

The order of the four images was randomized in each part of the survey. Respondents were asked to rank the images in terms of visual quality from best to worst. Since visual quality was never explicitly defined and the act of ranking images is a subjective process, respondents were asked to explain the criteria they used for their best image choices. The entire Visual Quality Survey is provided in Appendix A.

5.5.1.1 Survey Results

Below are the ranking results from the Visual Quality Survey for each benchmark scene.

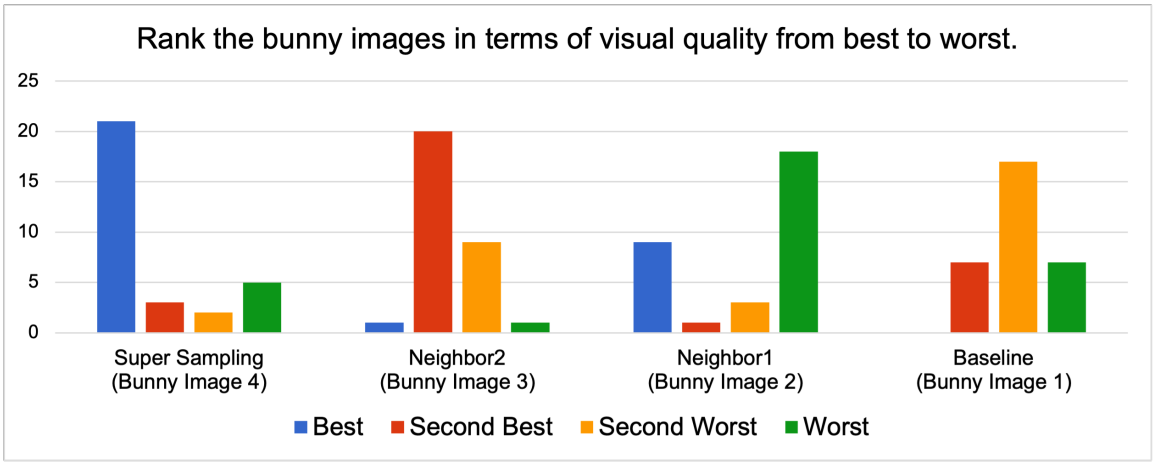


Figure 5.5: Bunny survey results

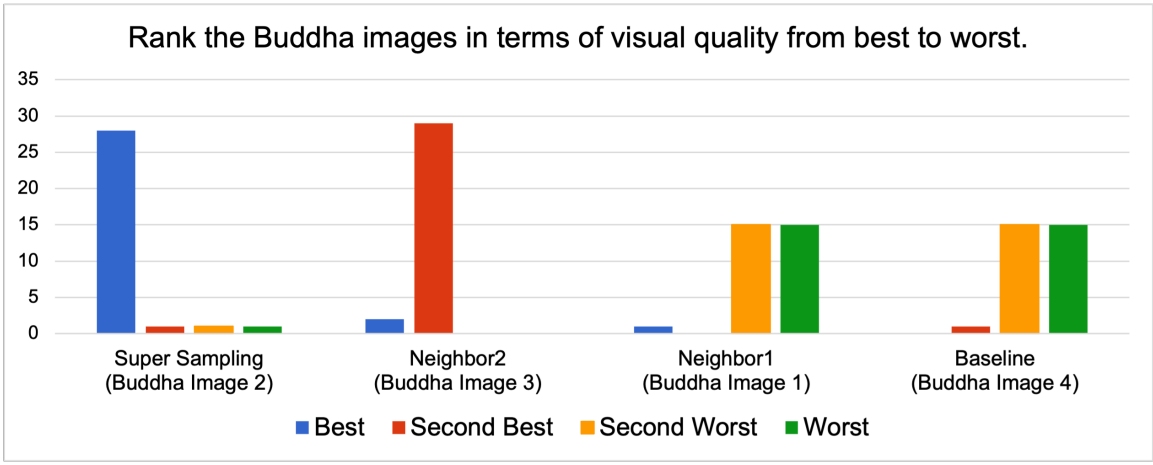


Figure 5.6: Buddha survey results

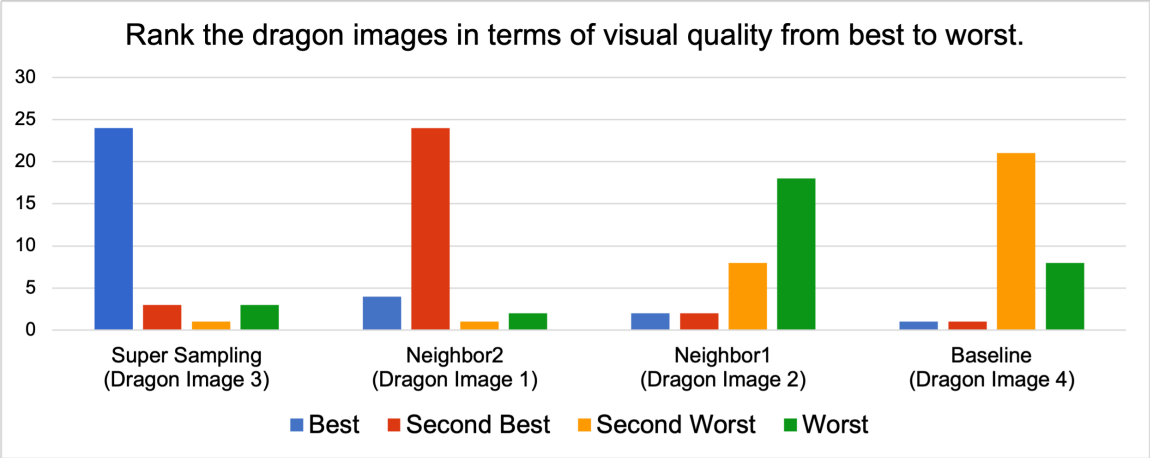


Figure 5.7: Dragon survey results

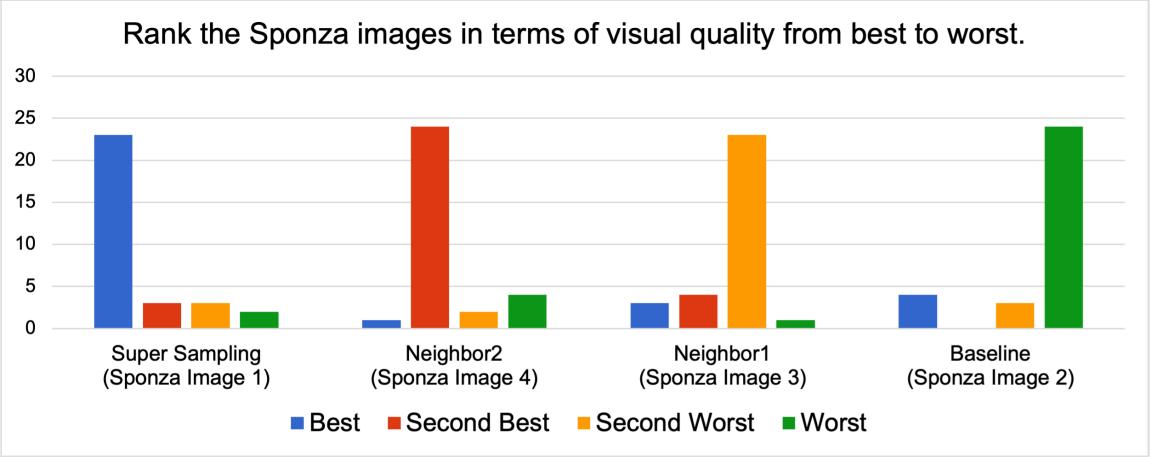


Figure 5.8: Sponza survey results

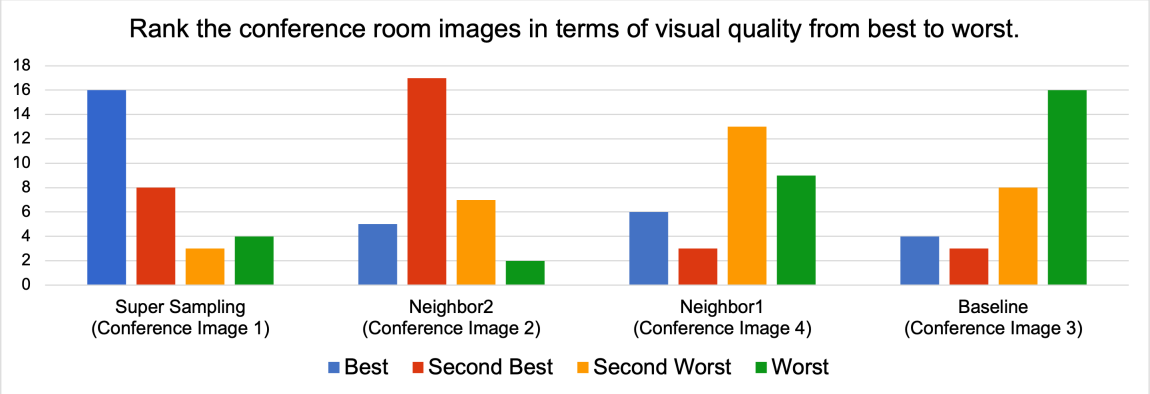


Figure 5.9: Conference survey results

5.5.1.2 Survey Analysis

The survey results indicate that super sampling produced the best images followed by the neighbor2 method. The baseline approach and the neighbor1 method received the lowest rankings. The baseline approach ranked worst for the Sponza and Conference benchmarks, and neighbor1 scored the worst for the Bunny and Dragon scenes. Both methods scored equally for second worst and worst on the Buddha scene.

After each ranking question, respondents were asked to explain the criteria or factors they used to choose the best images. Most of the survey participants stated that they judged images based on smoothness; they ranked images higher if the images appeared smoother than the others. Many focused on the blending of colors and the smoothness of the object outlines and edges. The people who did not choose super sampling as the best method often indicated that the more rough and grainy images looked more realistic, detailed, and textured. Some participants even indicated that the smoother and blended images appeared blurry.

The survey results validated some expectations and produced insight on design decisions. Since neighbor1 always ranked lower than neighbor2, it can be deduced that the neighbor1 approach did not consider enough voxel neighbors to accurately construct tangent planes and normals for each surface. A voxel radius of two seems to be a good choice since neighbor2 consistently scored higher than the baseline method. The results also show that super sampling scored the best on all benchmarks. However, some responses indicated that super sampling sometimes produced blurriness or removed sharpness from images. Future work could examine the effect of different super sampling rates on image quality. It would have also been interesting to learn about if people could tell a difference between a fully super sampled image versus a partially super sampled image by proximity.

Chapter 6

CONCLUSION

This thesis presented a new way to connect material information to Sparse Voxel DAGs during ray tracing. This work exemplifies several tradeoffs between memory, performance, and visual quality. Although the hash table material representation decreased build and render times and allowed the DAG to use less memory than a Moxel DAG, the hash table structure caused this work to use significantly more memory than the Moxel DAG implementation. To decrease memory requirements, surface normals for each voxel were approximated by calculating a tangent plane from surrounding voxels. This method greatly reduced memory and improved the visual quality of images, but the extra calculations increased the render times for most benchmark scenes. Lastly, super sampling was implemented and further improved visual quality at the cost of rendering time.

In comparison to the Moxel DAG implementation, the final implementation in this work overall decreased memory usage, decreased build times, improved visual quality, and increased render times. Although the rendering time of this work was slower, the added rendering time was smaller than the the build time saved. In other words, the total build and render times for this work were faster than the Moxel DAG build and render times.

6.1 Future Work

The ray tracing implementation used in this work only traces primary rays and shades with the Phong BRDF so future work could explore how to incorporate other features

efficiently with the Sparse Voxel DAG and hash table representation. Some features to examine include shadows and textures.

Similar to the findings in the Moxel DAG implementation, a majority of memory usage is due to the material data representation. For large resolution scenes and scenes with many more materials, it would be necessary to implement algorithms for material data compression.

Another direction of future work is tuning the different parameters of this implementation to find the best configurations. Some parameters of interest are the voxel radius for approximating surface normals, the voxels used for super sampling, and the number of rays to super sample with.

BIBLIOGRAPHY

[1] Brent Williams GitHub. `https://github.com/brentrwilliams`.

[2] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. *Eurographics*, 87(3), 1987.

[3] J. Baert, A. Lagae, and P. Dutré. Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th High-Performance Graphics Conference*. ACM, 2013.

[4] F.-C. Centre. Computer graphics, 2014. `http://www.fcc.chalmers.se/technologies/geo/computer-graphics/`.

[5] Chetvorno. Normal vectors on a curved surface, October 2019. `https://commons.wikimedia.org/w/index.php?curid=82974126`.

[6] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. ACM, 2009.

[7] B. Dado, T. R. Kol, P. Bauszat, J.-M. Thiery, and E. Eisemann. Geometry and attribute compression for voxel scenes. *Computer Graphics Forum*, 35(2), 2016.

[8] D. Dolonius. *Sparse Voxel DAGs for Shadows and for Geometry with Colors*. PhD thesis, Chalmers University of Technology, 2018.

[9] D. Dolonius, E. Sintorn, V. Kämpe, and U. Assarsson. Compressing color data for voxelized surface geometry. *IEEE Transactions on Visualization and Computer Graphics*, 25(2):1270–1282, 2017.

[10] I. Dunn. Lecture 6: Lighting.
https://calpoly-iandunn.github.io/csc473/lectures/06-lighting.

[11] I. Dunn and Z. Wood. Graphics program compendium.
https://graphicscompendium.com/index.html.

[12] J. Elseberg, D. Borrmann, and A. Nüchter. Efficient processing of large 3d point clouds. In *2011 XXIII International Symposium on Information, Communication and Automation Technologies*. IEEE, 2011.

[13] D. Eppstein. Z-curve, April 2008.
https://commons.wikimedia.org/wiki/File:Z-curve.svg.

[14] E. Ernerfeldt. Fitting a plane to noisy points in 3d, 2017. http://www.ilikebigbits.com/2017_09_25_plane_from_points_2.html.

[15] A. S. Glassner, editor. *An Introduction to Ray Tracing*. Elsevier, 1989.

[16] Henrik. Ray trace diagram, April 2008. Licensed under CC BY-SA 4.0 by https://creativecommons.org/licenses/by-sa/4.0/ with no changes.
https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg.

[17] V. Kämpe, E. Sintorn, and U. Assarsson. High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)*, 32(4):1–13, 2013.

[18] V. Kämpe, E. Sintorn, D. Dolonius, and U. Assarsson. Fast, memory-efficient construction of voxelized shadows. *IEEE Transactions on Visualization and Computer Graphics*, 22(10):2239–2248, 2016.

[19] S. Laine and T. Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011.

[20] M. McGuire. Computer graphics archive, July 2017.
https://casual-effects.com/data.

[21] C. E. V. Muniz and E. W. G. Clua. Finding surface normals from voxels. 2007.

[22] Ostfold University College Department for Information Technology. Some materials, April 2009. http://www.it.hiof.no/~borres/j3d/explain/light/p-materials.html.

[23] B. Smith. Phong components version 4, August 2006. Licensed under CC BY-SA 3.0 by https://creativecommons.org/licenses/by-sa/3.0/ with no changes. https://commons.wikimedia.org/wiki/File:Phong_components_version_4.png.

[24] A. J. Villanueva, F. Marton, and E. Gobbetti. Ssvdags: Symmetry-aware sparse voxel dags. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 2016.

[25] K.-Y. Whang, J.-W. Song, J.-W. Chang, J.-Y. Kim, W.-S. Cho, C.-M. Park, and I.-Y. Song. Octree-r: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, 1995.

[26] WhiteTimberwolf. Octree2, March 2010. Licensed under CC BY-SA 3.0 by https://creativecommons.org/licenses/by-sa/3.0/ with no changes. https://commons.wikimedia.org/wiki/File:Octree2.svg.

[27] B. Williams. Moxel dags: Connecting material information to high resolution sparse voxel dags. Master's thesis, California Polytechnic State University, San Luis Obispo, 2015.

[28] C. Wynn. An introduction to brdf-based lighting. *Nvidia Corporation*, 2000.

APPENDICES

Appendix A

VISUAL QUALITY SURVEY

For my thesis, I implemented different techniques that affect the quality of output images. I created this survey to see how people would perceive and compare the techniques. This survey asks you to rank images based on visual quality and to explain your choices. Thank you for your time!

(1/5) Rank the bunny images in terms of visual quality in the next question. Please zoom in to see the differences if necessary.

Bunny Image 1

Bunny Image 2

Bunny Image 3

Bunny Image 4

Rank the bunny images in terms of visual quality from best to worst.

	Best	Second Best	Second Worst	Worst
Bunny Image 1	○	○	○	○
Bunny Image 2	○	○	○	○
Bunny Image 3	○	○	○	○
Bunny Image 4	○	○	○	○

For the bunny image that you chose as "Best", what criteria did you use and/or what factors made this image better than the others?

Your answer

(2/5) Rank the Buddha images in terms of visual quality in the next question. Please zoom in to see the differences if necessary.

Buddha Image 1

Buddha Image 2

Buddha Image 3

Buddha Image 4

Rank the Buddha images in terms of visual quality from best to worst.

	Best	Second Best	Second Worst	Worst
Buddha Image 1	○	○	○	○
Buddha Image 2	○	○	○	○
Buddha Image 3	○	○	○	○
Buddha Image 4	○	○	○	○

For the Buddha image that you chose as "Best", what criteria did you use and/or what factors made this image better than the others?

Your answer

(3/5) Rank the dragon images in terms of visual quality in the next question. Please zoom in to see the differences if necessary.

Dragon Image 1

Dragon Image 2

Dragon Image 3

Dragon Image 4

Rank the dragon images in terms of visual quality from best to worst.

	Best	Second Best	Second Worst	Worst
Dragon Image 1	○	○	○	○
Dragon Image 2	○	○	○	○
Dragon Image 3	○	○	○	○
Dragon Image 4	○	○	○	○

For the dragon image that you chose as "Best", what criteria did you use and/or what factors made this image better than the others?

Your answer

(4/5) Rank the Sponza images in terms of visual quality in the next question. Please zoom in to see the differences if necessary.

Sponza Image 1

Sponza Image 2

Sponza Image 3

Sponza Image 4

Rank the Sponza images in terms of visual quality from best to worst.

	Best	Second Best	Second Worst	Worst
Sponza Image 1	○	○	○	○
Sponza Image 2	○	○	○	○
Sponza Image 3	○	○	○	○
Sponza Image 4	○	○	○	○

For the Sponza image that you chose as "Best", what criteria did you use and/or what factors made this image better than the others?

Your answer

(5/5) Rank the conference room images in terms of visual quality in the next question. Please zoom in to see the differences if necessary.

Conference Image 1

Conference Image 2

Conference Image 3

Conference Image 4

Rank the conference room images in terms of visual quality from best to worst.

	Best	Second Best	Second Worst	Worst
Conference Image 1	○	○	○	○
Conference Image 2	○	○	○	○
Conference Image 3	○	○	○	○
Conference Image 4	○	○	○	○

For the conference room image that you chose as "Best", what criteria did you use and/or what factors made this image better than the others?

Your answer

Additional comments or feedback on anything (optional)

Your answer

Submit