DELFT UNIVERSITY OF TECHNOLOGY

MASTERS' THESIS

Geometry and Attribute Compression for Voxel Scenes

Author:
Bas Dado

Supervisor:
Prof. Elmar Eisemann

This thesis was submitted
for the degree of Master of Science

in the

Computer Graphics & Visualization Group
Department of Intelligent Systems

December 9, 2015

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Department of Intelligent Systems

Master of Science

Geometry and Attribute Compression for Voxel Scenes

by Bas Dado

Voxel-based approaches are today's standard to encode volume data. Recently, directed acyclic graphs (DAGs) were successfully used for compression, but they are restricted to a single bit of (geometry) information per voxel. We present several methods to compress *arbitrary* data (e.g., colors, normals, or reflectance information). Our most successful method decouples geometry and voxel data via a novel mapping scheme, enabling us to apply the DAG principle on the geometry while compressing the voxel attributes using a specialized algorithm. This leads to a drastic memory reduction. Our method outperforms existing state-of-the-art techniques and is well-suited for GPU architectures, resulting in real-time performance on commodity hardware for colored scenes with up to 17 levels ($131,072^3$ resolution) treated in core.

Acknowledgements

This research was executed in collaboration with the Computer Graphics and Visualization (CG&V) group at the faculty of Electrical Engineering, Mathematics and Computer Science (EEMSC) of the Delft University of Technology (TU Delft). I would like to thank all members of this group for their input and support, but some members in particular.

First and foremost, I would like to thanks Timothy Kol for his day-to-day support throughout this project. He helped work out the ideas and algorithms used in this paper, and proved very important with the evaluation and testing. Secondly, I would like to thank Elmar Eisemann for his proposal to start this thesis, and for his very valuable input and ideas during our discussions. I would also like to thank Pablo Bauszat and Jean-Marc Thiery for our discussions, and for their ideas and contributions to this project. Finally, I would like to thank Thomas Höllt, Nicola Pezzotti and Renata Raidou for letting me work in their office, and Ruud de Jong and Bart Vastenhouw for providing me with a PC capable of shader debugging.

In addition, I'd like to thank my girlfriend, Linda, for her amazing support during my years at the TU Delft, and in particular during this final project. The biggest gratitude goes out to my parents for their endless, incredible support, and for giving me the opportunity to obtain a degree in engineering. Without you, none of this would have been possible.

Contents

Abstract i

Acknowledgements ii

1 Introduction 1

2 Background 3
2.1 Related Work . . . 3
2.2 Definitions . . . 4

3 Methods 6
3.1 Node count reduction . . . 6
3.1.1 Naive method . . . 6
3.1.2 Bittrees . . . 7
3.1.3 Geometry-material decoupling . . . 8
3.1.4 Variations . . . 9
3.2 Efficient tree storage . . . 10
3.2.1 Naive method . . . 10
3.2.2 Pointer and offset sizes per level . . . 10
3.2.3 Pointer entropy encoding . . . 11
3.2.4 Virtual nodes . . . 11
3.3 Data quantization . . . 11
3.3.1 Colors . . . 12
3.3.2 Normals . . . 12
3.3.3 Fixed point values . . . 14
3.4 Data compression . . . 14
3.4.1 Tight packing . . . 14
3.4.2 Repeated-block compression . . . 14
3.4.3 Bittree based compression . . . 15
3.4.4 Palette compression . . . 15

4 Implementation 18
4.1 DAG/Octree . . . 18
4.1.1 Construction . . . 18
4.1.2 Memory storage . . . 19
4.2 Renderer . . . 20

5 Results 22
5.1 Compression . . . 22
5.1.1 Efficient tree storage . . . 24
5.1.2 Data quantization . . . 25
5.1.3 Data compression . . . 26
5.2 Construction times . . . 29

5.3 Rendering performance . . . 30
5.4 Applications . . . 31

6 Conclusions **33**
6.0.1 Future work . . . 34

A Graph data tables **35**
A.1 Figure 5.1 . . . 35
A.1.1 4096 Quantized colors . . . 35
A.1.2 Full colors . . . 36
A.2 Figure 5.2 . . . 38
A.2.1 Geometry DAG . . . 38
A.2.2 Topology and offsets . . . 39
A.3 Figure 5.3 . . . 40
A.4 Figure 5.4 . . . 41
A.5 Figure 5.5 . . . 42
A.5.1 Lossy compression . . . 42
A.5.2 Lossless compression . . . 42
A.6 Figure 5.6 . . . 42

Bibliography **44**

Chapter 1

Introduction

FIGURE 1.1: Compressed voxelized scene at different levels of detail, rendered in real time using raytracing only. Our hierarchy encodes geometry and quantized colors, and is 17 levels deep, which corresponds to a voxel resolution of $131,072^3$. Despite containing 18.4 billion colored nodes, it is stored entirely on the GPU, requiring 7.63GB of memory using our compression schemes. Note that only at the scale shown in the bottom right image, the voxels become apparent.

Most 3D computer applications today make use of the standard rendering pipeline, in which scenes are represented as triangle meshes; collections of triangles that represent the surface of objects. Triangle meshes are rendered using triangle rasterization, a technique which determines for every pixel which triangle should be drawn on it (occupies the pixel and is closest). This means that, in practice, the number of triangles has a direct influence on the rendering performance, effectively limiting the amount of detail in a scene. Most application add detail to the triangle meshes by introducing textures, allowing for color variations on the surface of a single triangle.

In recent years, programmable shaders have allowed the use of textures for many more attributes, such as how light interacts with the surface (normal maps, bump maps, specularity maps), or even how the surface deforms (displacement maps [SKU08]). Although this makes scenes look more detailed, it increases the complexity of both the scene representation and how it is rendered.

Furthermore, the standard rendering pipeline does not allow for efficient ray-casting. Having this ability would enable advanced lighting effects, such as indirect illumination, (specular) scene reflections and ambient occlusion.

With the increase of complexity in virtual scenes and the rising importance of advanced lighting techniques, alternative representations, which are able to represent small details efficiently and enable efficient ray casting have received a renewed interest in computer graphics [LK10].

One alternative consists in voxels, which represent a scene in the form of a high-resolution grid. While voxels can represent complicated structures, the memory cost grows quickly. Fortunately, most scenes are sparse – i.e., many voxels are empty. For

instance, Figure 1.1 uses a grid of 2 quadrillion voxels ($8^{17} = 2\,251\,799\,813\,685\,248$), but 99.9994% are actually empty. This sparsity can be exploited using hierarchical representations, such as sparse voxel octrees (SVOs) [JT80; Mea82]. In addition to being more memory-efficient, SVOs can also be used to accelerate ray casting. However, they can only be moderately successful; a large volume like the previous example, still contains over 13 billion (i.e., around 8^{11}) filled voxels. Thus, specialized out-of-core and compression mechanisms are needed, such as those surveyed by [BR+14], but they cause additional performance costs.

Recently, Kämpe et al. [KSA13] used directed acyclic graphs (DAGs) to achieve high compression while keeping an in-core SVO representation with a single bit of information per leaf node. The idea is to merge equal subtrees, which is particularly successful if scenes exhibit repetition. Unfortunately, extending the information beyond a single bit (e.g., to store material properties) is challenging, as it would reduce the amount of equal subtrees drastically.

The goal of this thesis is to extent the DAG compression to allow for the inclusion of material information, while still maintaining significant compression rates. To this extent, we have developed and experimented with several compression schemes. The most effective scheme uses a mapping that decouples the topology from other voxel attributes. Hereby, we can apply the full DAG compression on geometry and include a special pointer reduction. The voxel attribute data can now be compressed using a specialized palette-based approach on quantized information, which greatly reduces the memory footprint.

With our method, a perceptually almost indistinguishable full-color voxel grid requires on average less than one byte per voxel (Figure 1.1). Additionally, attributes like normals or reflectance, can be compressed as well. Our representation has a low query cost, enabling complex rendering effects, such as specular reflections of the environment. Our approach displays, in full HD, a colored 8^{17}-voxel scene in real time on commodity hardware, keeping all data in core.

In order to explain our methods, Chapter 2 will provide a more detailed explanation of the SVO and DAG data-structures, as well as the most related work. Chapter 3 describes the four steps required for our compression algorithms: DAG conversion, efficient tree storage, attribute quantization, and attribute data compression. In Chapter 4, implementation details and caveats will be explained, as well as how we convert triangle meshes to voxel grids, and how we render our final data structure. Chapter 5 contains a discussion of the compression ratios and rendering performance of our algorithms. It also provides an insight in the quality of the quantization, as well as the time required by our current implementation for converting a triangle mesh to our compressed data structure. In Chapter 6, we will provide a final discussion on the effectiveness of our algorithms, and we will report future work.

Chapter 2

Background

This chapter focuses on the background of this thesis work. The most related literature is discussed in Section 2.1. The terms and definitions used throughout the rest of the paper are presented in Section 2.2.

2.1 Related Work

Here, we focus on the most related methods, but refer to other compression techniques, particularly for GPU-based volume rendering, to the recent survey by Rodríguez et al. [BR+14].

Streaming is a possibility to handle large data sets and recent approaches are able to adapt a reduced representation on the GPU taking into account the ray traversals through the voxel grid [EGG08; Cra+09]. Nonetheless, transfer and potential disk access make these methods less suited for high-performance applications. Here, it is advantageous to keep a full representation in GPU memory, for which compact data representation is of high importance.

Dense volume compression has received wide attention in several areas, e.g., in medical visualization [Gut+02]. These solutions exploit mostly local coherence in the data. While we also rely on this insight for data compression, such solutions are less suitable for sparse environments. In this context, besides SVOs [JT80; Mea82], perfect spatial hashing can render a voxel dataset by means of a hash and offset tables [LH06]. While these solutions support efficient random access, exploiting sparsity alone is insufficient to compress high-resolution scenes.

Efficient sparse voxel octrees (ESVOs) observe that scene geometry can generally be well represented using a contour encoding [LK11]. This, combined with block-based compression based on DXT1, allows for reasonably efficient storage of SVOs. Nonetheless, due to subtree culling where the contour error is below a certain threshold, this representation does not retain the original precision of the stored attributes (e.g., color). While it is possible to account for colors when a subtree is culled, this choice reduces the compression effectiveness drastically.

Recently, Kämpe et al. observed that besides sparsity, geometric redundancy in binary voxel scenes is common, and they proposed a scheme to merge equal subtrees in an SVO, resulting in a compressed directed acyclic graph (DAG) [KSA13]. The compression rates are significant and the method even found applications in shadow mapping [Sin+14; KSA15]. Nonetheless, the employed node pointers to encode the structure of the DAG can become a critical bottleneck.

Pointerless SVOs [SK06] are well-suited for offline storage, but have slow runtime access. While some efficient suggestions were made [LK11; LH07], these methods are typically not applicable to the DAG, since they are usually based on the assumption that pointers can be replaced by small offsets. In case of the DAG, this can lead to large

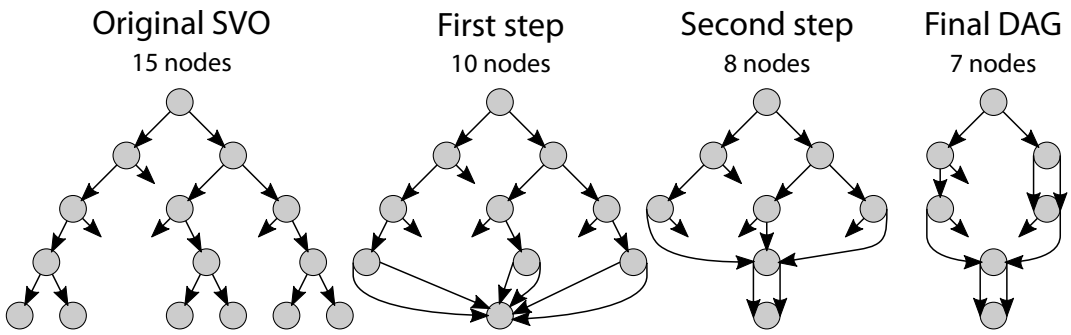


FIGURE 2.1: Illustration of the DAG conversion algorithm on a binary tree. Dangling pointers illustrate empty nodes. In a bottom-up fashion, all equal nodes are merged.

values, as the node's children are no longer in order, but potentially scattered across different subtrees.

Increasing the data per voxel reduces the probability of equal subtrees, making DAGs unsuitable for colored scenes. For attribute compression, specialized algorithms exists for textures [SAM05; Nys+12], as well as colors (e.g., via an effective quantization as in [Xia97]) or normals (e.g., via a octahedron-normal vectors (ONVs) as in [Mey+10]), for which a careful quantization is necessary [Cig+14].

Recently, Williams proposed a mapping based on storing the number of empty nodes in a subtree to connect material information to high resolution sparse voxel DAGs [Wil15]. However, since by far the largest part of a sparse scene is empty, storing the amount of empty nodes in a subtree can require very big integers, and thus a lot of memory. For the scene in Figure 1.1, for example, one would require at least 48-bit integers to store the number of empty nodes correctly. Furthermore, attribute compression is not explored.

2.2 Definitions

A voxel scene is a cubical 3D grid of size N^3, with N a power of two. Each voxel is either empty or contains some information, such as a bit indicating the presence of geometry, normals, colors or other attribute data. SVOs encode these grids by grouping homogeneous regions; each node stores an 8-bit *childmask* denoting for every child node if it exists – i.e., is not empty. A *child pointer* points to the children, which are ordered in memory. Hence, 8 bits are needed for the childmask, plus 32-bit for the child pointer. Furthermore, for level-of-detail rendering, parent nodes usually contain data representing that of the children (e.g., an average color). If only geometry is encoded, testing the presence of a child pointer is sufficient and no data entries are needed.

The DAG algorithm is an elegant method to exploit redundancy in the SVO. It forms the basis for all our proposed compression schemes. For ease of illustration, Figure 2.1 uses a binary tree, but the extension to more children is straightforward. On the left, a sparse binary tree is shown. Dangling pointers refer to empty child nodes without geometry. The DAG is constructed in a greedy bottom-up fashion; subtrees (starting with the leaf nodes at the lowest level) are compared and identical ones are merged by changing the parent pointers to point to a single common subtree. The final DAG exhibits significantly less nodes (Figure 2.1 right).

One disadvantage of the DAG in comparison to an SVO, is that pointers need to be stored for *all* children, because children can no longer be grouped consecutively in memory (in which case, a single pointer to the first child is sufficient). In practice, the 40 bits per node in an SVO (8-bit childmask and a 32-bit pointer), become around $8 + 4 \times 32 = 136$ bits in a DAG – an octree node on average has about four children, when voxelizing surface models. The high gain of the DAG stems from the compression at low levels in the tree. In a typical SVO, these levels are the bottleneck, containing by far the most nodes. A DAG has at most 256 leaf-nodes; the number of unique combinations. For higher levels, the number of combinations increase, which reduces the number of possible merging operations. This also reflects the difficulty that arises when trying to merge nodes containing data.

Chapter 3

Methods

It is possible to distinguish several steps for our compression algorithm aimed at voxel scenes. The first step is converting from an SVO to a DAG. We distinguish several schemes to maximize the number of equal subtrees, and in extension the effectiveness of the DAG conversion. The second step consists of reducing the memory needed to store the child pointers. The third step revolves around quantizing the voxel attributes, which reduces the entropy and thus allows for more efficient storage. Although quantization is by definition lossy, differences can be made sufficiently small so as not be distinguishable by humans. The last step compresses voxel attributes further, by exploiting spatial and structural coherence, and is required for our geometry-material decoupling to be effective.

3.1 Node count reduction

The DAG conversion algorithm as proposed by Kämpe et al. does not allow for storing attribute data. Nonetheless, we aim to use the DAG compression as the basis of our algorithm. In this section, we propose several modification to the DAG algorithm that enable the storage of attributes.

3.1.1 Naive method

A naive approach to material SVO compression, is to simply extend the standard DAG compression scheme to incorporate materials. In the standard DAG compression, nodes are merged bottom-up if they have identical childmasks and pointers. This condition

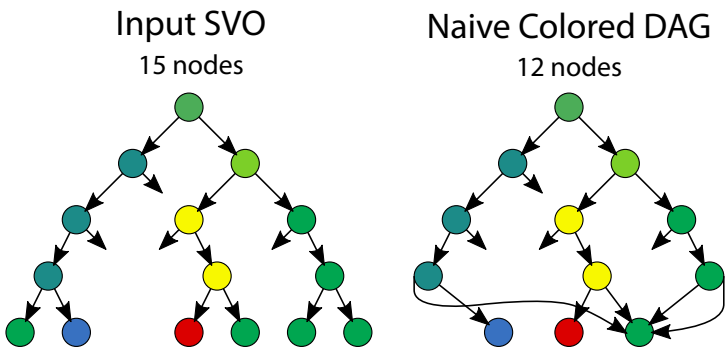


FIGURE 3.1: Illustration of the DAG conversion algorithm when attribute information, such as colors, are included. This is the naive way to include colors in the DAG compression algorithm.

can be extended such that nodes can only be merged if they also have the same materials.

However, this additional condition has a significant negative influence on the amount of merge opportunities, especially for scenes with diverse materials. An example of this method is shown in Figure 3.1. In this example, only $3/15$ nodes could be removed, compared to the $8/15$ nodes for the geometry DAG. Quantization techniques (Section 3.3) help but are not sufficient.

3.1.2 Bittrees

To increase the number of merge opportunities compared to the naive approach, one could opt to reduce the material information per node. This can be achieved by splitting the material information into several small parts (e.g. bits). Each part is stored in a seperate SVO, or *bittree*. It is reasonable to assume that there is a big overlap between these trees. Modifying the DAG algorithm to only store the same subtree once for all bittrees can exploit this assumption.

If the original geometry DAG is stored sperately, we can ensure that only the voxels that contain geometry are visited. This allows us to store arbitrary attribute information for the position that do not contain geometry. Since the attributes in these positions are essentially undefined, we can merge subtrees in bittrees if their only differences occur in positions that do not contain geometry.

Allowing this, however, substantially increases the complexity of the DAG conversion algorithm. The original DAG algorithm runs in $O(N \log N)$ and is guaranteed to give an optimal solution. However, when there are multiple different nodes that can be merged, as is the case when part of the scene is regarded as undefined, the problem becomes much harder.

It is no longer possible to find a sorting of the nodes such that all nodes that should be merged are adjacent. The trivial way to find all viable merges for a single node, is to check all other nodes. This would increase the complexity to $O(N^2)$. Low level trees already require millions of nodes, rendering an algorithm that runs in $O(N^2)$ unfeasible. We have implemented an acceleration tree structure with 8 levels, one for each child. Branching is based on the value of the child, which is either a pointer with attribute data or undefined. Pseudocode for the algorithm to find all viable merge opportunities is given in Algorithm Algorithm 1.

Secondly, a choice needs to be made which of the viable nodes should be merged. Merging with one node might prevent the possibility to merge correctly with another, and this selection could influence the efficiency of the DAG algorithm higher up in the tree. This is a hard combinatorial problem. Our solution is a greedy algorithm, that first merges nodes with the most parents (as these have a higher probability of leading to more merge opportunities higher in the tree).

Nonetheless, even with the acceleration tree, the algorithm is not feasible for large trees, and the results for smaller trees do not encourage further optimization. Therefore, despite the algorithm being implemented, we do not use it in practice.

Instead, we have implemented approximations that are much faster to execute and lead to similar results: First of all, we fill the nodes above the leaf level with children which have their bit value not set. This ensures that these nodes always have 8 children, making the attributes of these children the only difference between them nodes. This ensures that there can only be 256 different nodes in the level above the leafs, as opposed to $256 \cdot 256 = 65563$. Secondly, we cull subtrees for which all nodes that exist have the same bit value (e.g. all 1 or all 0). Finally, we store the bit value of each node

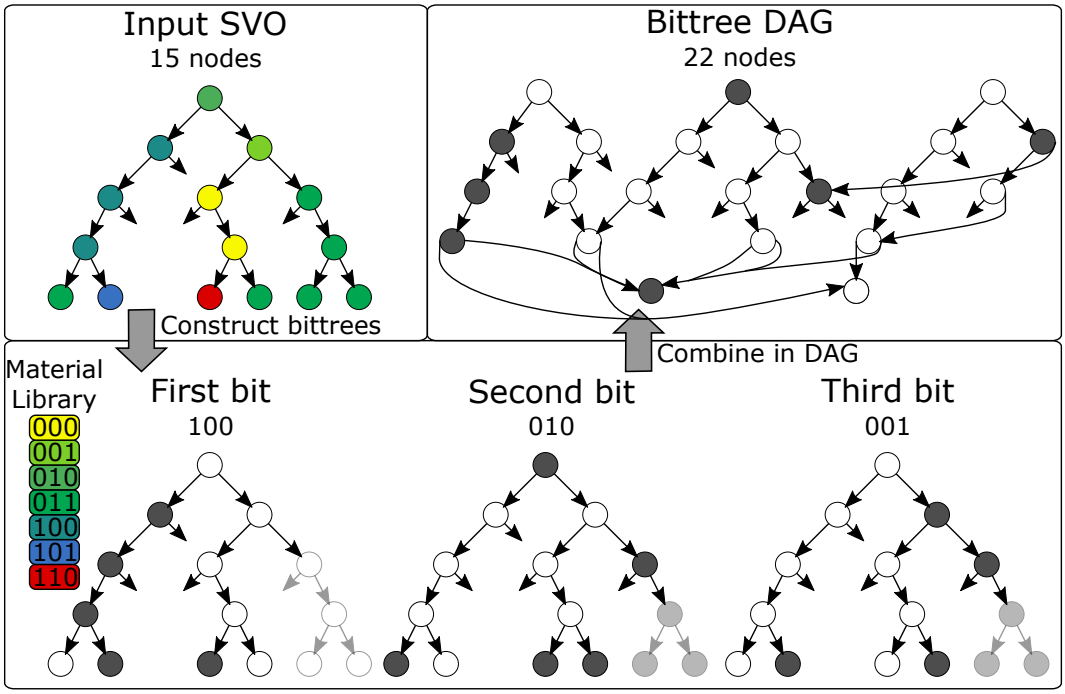


FIGURE 3.2: Illustration of the bittrees technique. A standard geometry DAG needs to be stored in addition to the bittree DAG in order to render correctly. Because this example is relatively small, the bittree DAG currently does not reduce the number of nodes, but this could happen for a bigger SVO.

in their parent. This allows for merging nodes that have a different bit value, but the same children and topology. It is also convenient for storage, as nodes need to be byte-aligned in memory. The current implementation for bittrees is illustrated in Figure 3.2.

3.1.3 Geometry-material decoupling

A final approach to simplify the SVO structure is to decouple material and geometry information. This allows for using a separate compression scheme specialized in compressing either geometry or attribute information.

To achieve this decoupling, we assign *data indices* to all nodes in the initial SVO in a depth-first order (numbers in the nodes of Figure 3.3). Next, for every child pointer, we store an *offset* integer, such that adding all offsets along the way from the root to a node results in the node's data index (numbers next to the edges in Figure 3.3, right). Since the first offset is always $+1$, it is stored implicitly. In this representation, the DAG algorithm becomes significantly more efficient then storing the materials directly. In fact, we obtain exactly the same compression as for geometry only, as depth-first indexing automatically leads to identical offsets in identical subtrees. Still, our mapping does introduce an overhead in the form of a 32-bit offset for every pointer. However, as explained in Section 3.2, the number of bits used for these offsets can be easily reduced in practice.

Note that this method requires the storage of the attribute of all nodes in the original SVO in a separate *node data table*. However, due to the data indices being assigned

Algorithm 1 Pseudocode to traverse the acceleration tree used during bittree construction. It is used to find all nodes viable for merging with a given node, taking undefined space into account.

```
1:  function TRAVERSE(children[8])                    ▷ children from the original node in the DAG
2:      ▷ Start at the root of the acceleration tree
3:      return TRAVERSE(children, root, 0)
4:  end function
5:  function TRAVERSE(children[8], node, depth)
6:      child ← children[depth]
7:      if child = undefined then                     ▷ child lies in region that does not contain geometry
8:          for all childNode in GETCHILDREN(node) do
9:              yield TRAVERSE(children, childNode, depth + 1)
10:         end for
11:     else
12:         ▷ findChild returns the child from the acceleration tree node that represents the DAG node
13:         childRepresentative ← FINDCHILD(node, child)
14:         undefinedRepresentative ← FINDCHILD(node, undefined)
15:         yield TRAVERSE(children, childRepresentative, depth + 1))
16:         yield TRAVERSE(children, undefinedRepresentative, depth + 1))
17:     end if
18: end function
```

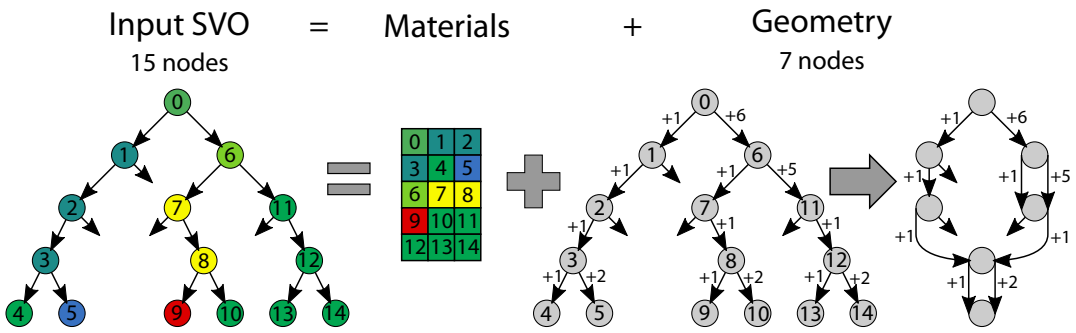


FIGURE 3.3: Illustration of the geometry material decoupling technique. Each node gets assigned an index, which is reconstructed by summing the offsets from the root to the node. This allows for full compression of the geometry DAG. A separate table stores the materials for each node.

in a depth-first manner, the node data table preserves spatial coherence. This can be exploited by a specialized compression scheme that would not be possible if the attribute data was stored directly in the tree structure, as explained in Section 3.4. Additional material quantization (Section 3.3) can also be utilized to reduce the memory footprint. By only assigning indices up to a user-defined level in the tree, we can reduce the resolution of the attribute data, without losing geometry detail. This does not influence the DAG compression and is a useful tool to balance between quality and memory usage.

3.1.4 Variations

We have experimented with several variations on the methods described above. The first variation we tried was to store differences in materials, instead of storing materials directly. Since the materials in a child node are often similar to those in a parent node, these difference should be relatively small and predictable. Unfortunately, many nodes did not follow this pattern due to high frequency content in the textures. This was

especially apparent near the leaf nodes, where compression is most crucial. In none of our experiments did storing differences result in a better compression.

We have also experimented with bittrees that only store information in the leaf nodes, meaning that they are essentially equal to geometry trees. These allowed for some added compression compared to standard bittrees. Nonetheless, the added compression did not overcome the loss of the level-of-detail information present in the SVO.

A final method we explored revolves around merging a naive DAG such that locations without geometry are allowed to contain any value. The algorithm for this is similar to the one described in Section 3.1.2. Note that due to the algorithms complexity, evaluation on big datasets was not attainable. For smaller datasets, it gave a slight improvement over the naive method, but was still outperformed by our geometry-material decoupling.

3.2 Efficient tree storage

This section presents several methods we have used to store DAG nodes as efficiently as possible. We can distinguish four main methods: the naive approach (which is equal to the original DAG paper), selecting pointer and offset sizes per level in the tree, using entropy encoding to exploit repeating pointers, and using virtual nodes to exploit that low levels in the tree are not compressed.

3.2.1 Naive method

The naive method is equal to what was presented in [KSA13]. Every pointer (as well as the child mask) is stored in a word-aligned manner. This entails that both the child-mask (of which only 8 bits are used), as well as the child pointers are stored as 32-bit variables.

Having all data stored in a word-aligned manner is beneficial for performance, and this method makes it trivial to find the location of a pointer in memory.

3.2.2 Pointer and offset sizes per level

A DAG usually contains few nodes in the lowest levels of the tree (i.e., near the leafs), even though many pointers to these nodes are required. We exploit this by sorting the nodes on the level they are in. We then store a pointer to the first node of each level. This allows general child pointers to be replaced by pointers within each level. Taking the $\log_2$ of the memory used by each level in the tree, we can calculate the minimal amount of bits needed to store pointers to each level. For performance and simplicity reasons, these numbers are rounded up to bytes, so that we can use byte-precise pointer sizes. Experiments show that this makes little difference for the final size of the tree.

A similar technique can be applied to the offsets that are used for geometry-material decoupling. Here the technique is even more effective as these offsets get smaller as the node level increases. As a consequence, the levels with the most nodes, which are located a few levels above the leaf nodes, have relatively small offset sizes.

Using this technique does imply a small performance cost, as information is no longer word-aligned, and additional fetches are required to find the pointer and offset sizes for the current level. However we can now store the child mask of each node using 1 byte.

3.2.3 Pointer entropy encoding

In a DAG, some nodes are reused many times (i.e. have many parents), while others are used only once. For example, we found that for DAG representing a typical game scene, 70% of pointers to the biggest level in the point to the first 10% of nodes in that level. This property can be exploited using entropy encoding. The main idea is to use less bits for pointers to nodes that are used often. We implemented this using a 1- or 2-bit mask preceding each pointer which indicates its size.

To make sure that often-used nodes require smaller pointers, we sort the nodes per level on how many parents they have. As with pointer sizes per level, we replace pointers by the offsets within each level. This ensures that the most used node in each level will have 0 as its pointer, the second most used will have a pointer that is equal to the size of the most used node in bytes, etc. This means that we only need 1 byte (containing the mask and some 0's) to encode a pointer to this most used node. For each pointer, the minimum required size is used.

Since nodes have a size bigger than one, node pointers are not consecutive numbers. To store them more efficiently, a lookup table is be created for each level. Nodes are added to the lookup table in descending order of number of parents. We stop adding nodes to the lookup table if one of the following conditions hold:

- The current node has only 1 parent.
- Storing the index of the current node requires as many bytes as a standard pointer to this level.

3.2.4 Virtual nodes

The main difference between DAG and SVO nodes, is that DAG nodes need to store a pointer to each of their children, whereas SVO nodes only need to store a single pointer to the first of their children. This is required because a DAG reuses nodes, so a node's children cannot always be stored in order. However, many nodes are still used only once, and these nodes can be ordered in the same way they would be in an SVO.

To allow for using a single pointer per DAG node, we introduce *virtual nodes*. We order all nodes similarly to how they would appear in a standard SVO, making sure that the children of a node are stored in order next to each other. Whenever a node is reused, we replace the node by a virtual node, which is a pointer to the first occurrence of this node. Near the leaf nodes, many nodes are reused, so that using standard DAG storage, where the pointers are stored directly in the nodes, becomes beneficial. We calculate the memory requirements of each level with and without enabling virtual nodes, and use the method with the smallest memory requirement for each individual level.

3.3 Data quantization

This section deals with quantization techniques. Quantization is the problem of replacing some set of values by a representative smaller set, where the goal is to minimize both the number of samples and the error between the original values and their quantized counterparts. Quantizing the materials in a scene not only allows us to use fewer bits for each node, but also improves the effectiveness of other compression algorithms. To this extent, we have implemented different techniques for colors, normals and fixed point values. Color quantization is based on a technique proposed by

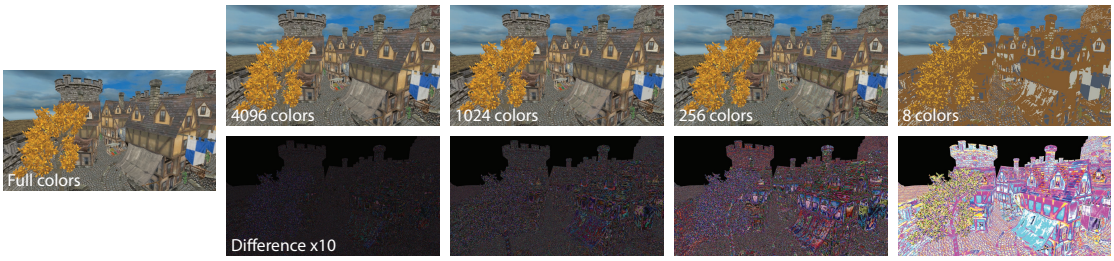


FIGURE 3.4: Color quantization. Top row shows quantization result, bottom row shows the difference to the ground truth, multiplied by 10. At 4096 colors, the result is perceptually almost indistinguishable from the ground truth, as demonstrated by the difference image, which seems nearly black even when multiplied by 10. For 1024 colors, the difference is noticeable, but high-quality results are still obtained. For 256 colors, it is clearly visible that the colors have been quantized, but the results are not uncomfortable. Only for 8 colors do we obtain unusable results.

Xiang [Xia97]. Normal quantization is based on octrahedral normal compression, as proposed by Meyer et al. [Mey+10].

3.3.1 Colors

Although humans can distinguish many different colors, often only a relatively small subset of these occur in a single scene. In addition, the RGB space is not perceptually uniform, leading to subsets being indistinguishable to humans. This allows for effective color quantization.

Our clustering method is based on a method proposed by Gonzalez et al. [Gon85], which uses the minimal maximum intercluster distance as an error metric. Intuitively, it attempts find clusters in which the entries are as close to each other as possible. Finding optimal clusters is NP-hard, but the algorithm proposed by Gonzalez et al. [Gon85] finds a 2-approximation in $O(kn)$ time, where k is the number of clusters and n the number of values.

Xiang [Xia97] first proposed using said algorithm on color quantization, by applying it to a scaled RGB space. We extend it by using the CIELAB color space [Cie], which is designed to be perceptually uniform. This means that the Euclidean distance between two points in this space can be used as a metric for the perceptual difference between the two colors these points represent.

The original algorithm required the user to define a number of clusters k. We added an alternative stop condition in the form of a maximum error ε. Effectively, this means we stop splitting clusters if no cluster has a color that is further than a distance ε to its cluster representative. Since CIELAB is perceptually uniform, this allows the user to select a maximum perceptual difference between the original and quantized colors. Picking a sufficiently small distance (e.g, 1.3) thus leads to a perceptually indistinguishable quantization.

3.3.2 Normals

As opposed to colors, a large subset of all possible normals usually appears within a scene. For example, if there is a single sphere in the scene, all possible normals are represented. Therefore, using an input specific quantization algorithm is not worthwhile for normals. Instead, we opt to use a standard method for quantization and storing of unit vectors, in the form of octahedron normal vectors (ONVs). ONVs were

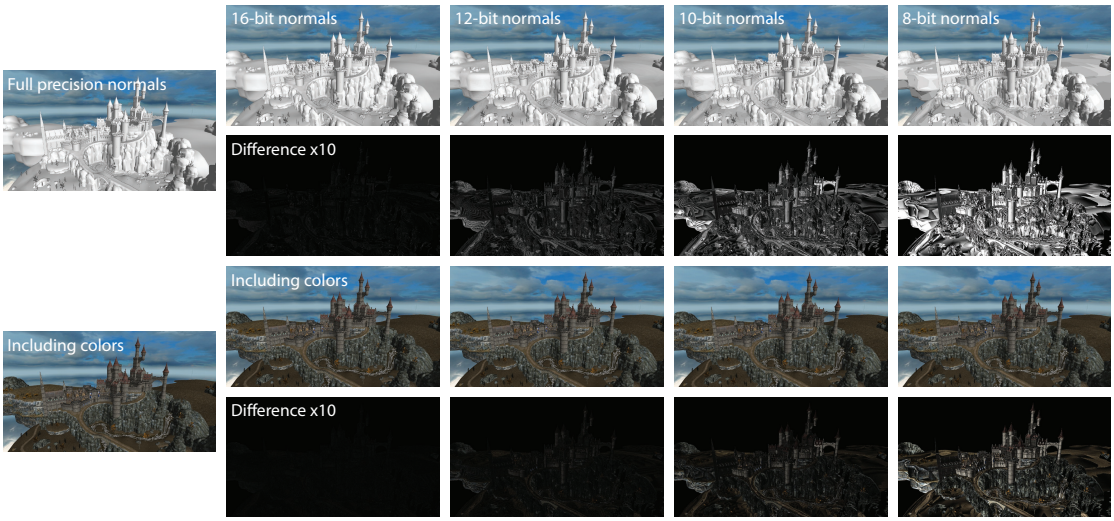


FIGURE 3.5: Normal quantization. Top row shows shaded result for quantized normals, with the difference to the ground truth below, multiplied by 10. Third row shows the shaded result for quantized normals and full colors, with the difference images below. 16-bit normals produce results perceptually indistinguishable, and even 12-bit normals exhibit high-quality results, especially when colors are included. For 10-bit normals, the difference becomes noticeable, and for 8-bit normals it is obvious, yet the colored result still looks reasonable.

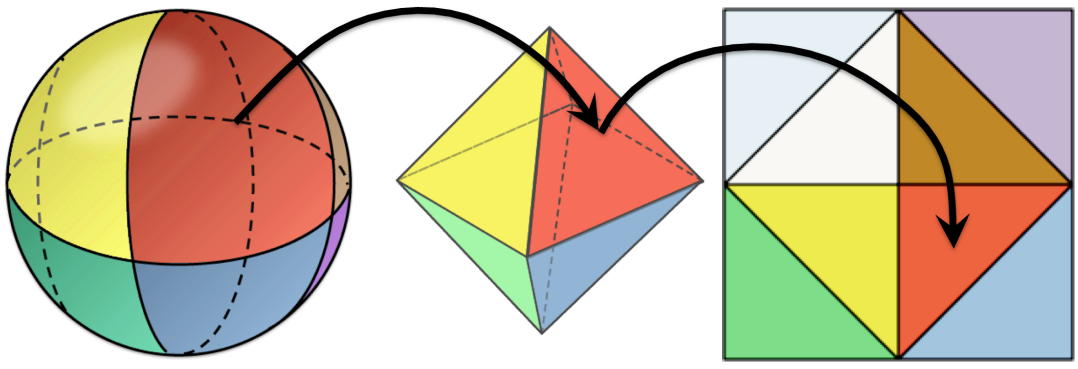


FIGURE 3.6: Illustration of how octahedron normal vectors map a 2D unwrapping of an octahedron to a sphere.

first proposed by Meyer et al. [Mey+10], and a recent survey by Cigolle et al. [Cig+14] confirmed their effectiveness. ONVs encode unit vectors by mapping them to an octahedron using the L^1-norm (Manhattan distance). The location on the octahedron's surface is stored as a (u, v) coordinate on an unwrapping of the octahedron to a square, as shown in Figure 3.6. This method distributes samples on the sphere in an almost uniform manner, where the maximum angle between a sample and the correct normal was shown by Meyer et al. to be:

$$\Delta_{\max} = \arccos \sqrt{\frac{2}{2 + 9\varepsilon^2}} = \frac{\sqrt{2}}{2} 3\varepsilon + O(\epsilon^3). \tag{3.1}$$

Where ε is the sample spacing.

3.3.3 Fixed point values

Some values, such as the opacity or reflectivity of a voxel, are usually in the range $[0, 1)$. We quantize these values by storing them as an unsigned integer:

$$v_q = \lfloor v \cdot v_{\max} \rfloor \tag{3.2}$$

Where v is the original value, v_q is the quantized value, and $v_{\max}$ is the number of samples. Usually $v_{\max}$ is a power of two. This allows for full utilization of $\log_2 v_{\max}$ number of bits.

3.4 Data compression

This section deals with compression of the node data table, which is required to store materials after the geometry-material decoupling. The compression techniques proposed in this chapter exploit several properties of this table. First, there are generally relatively few unique attributes/materials in a scene, especially after quantization. Second, we assume a high spatial coherence to be present in a regular scene (remember that spatial coherence in the original scene is mostly preserved in the node data table).

We define N to be the length of the node data table and M to be the number of unique materials.

3.4.1 Tight packing

Tight packing is a trivial method to exploit that there are usually few unique values. To make sure that we can index all unique values by subsequent indices, we create an array that contains them exactly once, the *material library*. We then replace all materials by pointers to this material library. These pointers thus require only $\log_2 M$ bits.

3.4.2 Repeated-block compression

Repeated-block compression exploits repeated patterns in the scene. The algorithm works by splitting the node data table into blocks of a fixed size P. All unique blocks are stored in a dataset, the *block library*. The node data table is now stored as $\lceil N/P \rceil$ indices to the block library. For additional compression, tight packing can be used on the block library.

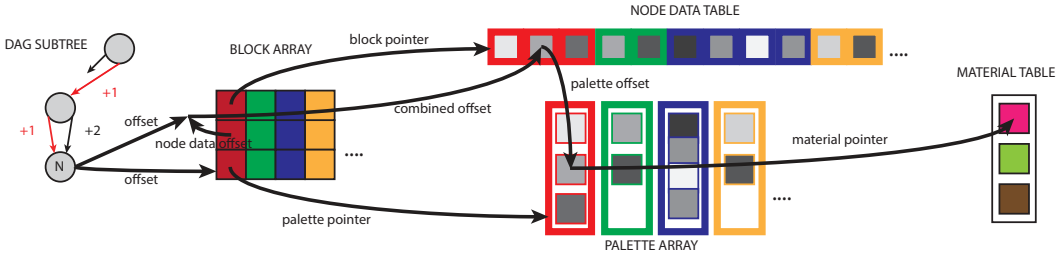


FIGURE 3.7: Illustration of the palette compression data structure and how an entry can be queried.

3.4.3 Bittree based compression

For standard bittrees, as explained in Section 3.1.2, the geometry information is mostly duplicated throughout the trees. This is inefficient. When the geometry and materials are decoupled, however, this redundancy is not present. We therefore hypothesize that using bittrees to compress the node data table will result in a smaller memory footprint. For our implementation of bittrees for the node data table, information is only stored in the leaf-nodes, where the existence of a leaf nodes indicates a 1 or a 0 for the bit represented by this position. To calculate which child of a node at level L in some bittree represents the value corresponding to some index i in the node data table, we use the bits of i located at position:

$$(D-L)\cdot 3 \text{ for the } x \text{ coordinate of the child.}$$
$$(D-L)\cdot 3+1 \text{ for the } y \text{ coordinate of the child.}$$
$$(D-L)\cdot 3+2 \text{ for the } z \text{ coordinate of the child.}$$

Here, D is the depth of the tree, defined as:

$$D = \log_2(N/8) = \frac{\log_2 N}{\log_2 8} = \frac{\log_2 N}{3} \tag{3.3}$$

This preserves some spatial coherency, and is relatively easy to calculate.

We have also experimented with using a dense material DAG to store the node data table, but this was not worthwhile.

3.4.4 Palette compression

This method aims at exploiting spatial coherence in the scene. We do this by finding large variably-sized consecutive blocks of data with few unique materials. For each of these blocks, we create a *palette* containing its unique materials. The data is stored as indices into this palette. As the palette is small, these indices use very few bits. Note that the palette itself is in turn stored as indices to the material library, using tight packing (Section 3.4.1) for efficiency.

In order to query random entries of the node data table, we use a set of *headers*. These headers contain, for each block, the index of the start of the block in the original node data table, a pointer to the palette for this block, a pointer to the start of this block in the compressed node data table, and the size of entries in the current block (proportional to the $\log_2$ of the palette size). To find the block in which some entry resides, a

binary search is applied on the headers, using the node table indices ($O(\log N)$). An image summarizing the storage and fetching from our palette compression data structure is shown in Figure 3.7.

The problem is now to find a set of blocks that covers the whole texture and uses the smallest amount of memory possible. This, however, is a hard combinatorial problem, and finding an exact (optimal) solution is not feasible. Instead, we use a two-step algorithm to find an approximation of the optimal solution. Both steps are greedy on the average size per entry in a block (including palette and header), but build the blocks to compare in a different manner.

The first step considers all maximum-sized blocks that can be made with palettes of size 1, 2, 4, and 8 respectively, corresponding to 0, 1, 2 and 3 bits per entry. A block is *maximum-sized* if its current palette contains the maximum number of allowed materials, and adding another entry to the block (either left or right) would require adding a material to the palette. *Finalizing* a block means that this block will be reserved for the final compressed data structure. Therefore, once a section of the node data table is covered by a finalized block, it cannot be claimed by another block later in the algorithms execution.

The algorithm starts by finding all maximum-sized blocks with 1 material. These are finalized in order of descending size, until the following condition no longer holds, with B the number of bits per entry (0 for blocks with 1 material), and N the number of entries in the block:

$$N \cdot B + [\text{header size}] + 2^B \cdot [\text{material size}] < N \cdot (B + 1). \tag{3.4}$$

It then continues combining previously found, non-finalized, maximum-sized blocks with 1 material to find all maximum-sized blocks with 2 materials. These are, again, finalized in order of descending size until the condition in **??** (with $B = 1$) no longer holds, skipping blocks that overlap an already finalized block. The algorithm continues this pattern for palettes of sizes 4 and 8, corresponding with $B = 2$ and $B = 3$ bits.

For added clarity, pseudocode that resolves to the same solution is presented in the function PROCESSSTEPONE of Algorithm Algorithm 2. Note that the actual implementation combines found blocks to find bigger maximum-sized blocks, in parallel on multiple CPU cores. This is much faster than implementing the algorithm in a recursive manner as shown in the pseudocode.

The second step is more rigorous. It starts at the first entry that is not part of a finalized block in the node data table, and finds all maximum sized blocks starting at that position using 0-8 bits per entry (palette sizes up to 256). Note that these blocks cannot overlap finalized sections of the node data table. For each of these blocks, it calculates the average size per entry:

$$\text{Average size per entry} = \frac{N \cdot B + [\text{header size}] + 2^B \cdot [\text{material size}]}{N}. \tag{3.5}$$

If the average size per entry is bigger than the size of a material, the algorithm is allowed to make blocks that use the full material library (and thus do not have a palette). These blocks do not require a block-specific palette to be stored, as they use the main material library, leading to less overhead. The block with the lowest average size per entry is finalized. This process is repeated until the entire node data table is covered.

Pseudocode of this step is shown in the function PROCESSSTEPTWO of Algorithm Algorithm 2.

Algorithm 2 Pseudocode for step 1 and 2 of the palette compression algorithm. Note that the actual implementation is more efficient, but has the same results.

function PROCESSSTEPONE(data$[i_1 \cdots i_2]$)
  if $i_1 \geq i_2$ **then return** ▷ Empty block
  end if
  $B \leftarrow 0$
  while $B \leq 3$ **do**
    data$[j_1 \cdots j_2] \leftarrow$ largest block representable with 2^n colors in data$[i_1 \cdots i_2]$
    ▷ Heuristically test memory cost to decide if larger palette is beneficial
    $N \leftarrow j_2 - j_1$
    if $N \cdot B$+paletteMemoryOverhead(2^B) $\leq N \cdot (B + 1)$ **then**
      ▷ Keep this block and apply algorithm recursively
      FINALIZE(data$[j_1 \cdots j_2]$) ▷ Finalize creates the palette and entries
      PROCESSSTEPONE(data$[i_1 \cdots j_1 - 1]$)
      PROCESSSTEPONE(data$[j_2 + 1 \cdots i_2]$)
      return
    else ▷ Expand palette, hence, increase B
      B++
    end if
  end while
  ▷ If no viable block is claimed this way, use a more rigorous approach: step two
  return PROCESSSTEPTWO(data$[i_1 \cdots i_2]$)
end function
function PROCESSSTEPTWO(data$[i_1 \cdots i_2]$)
  $s \leftarrow i_1$
  while $i_1 \leq i_2$ **do**
    for $B = \{0, 1, \cdots, 8\}$ **do**
      ▷ Find the maximum-sized block with entries that take B bits
      $potentialBlocks[B] \leftarrow$ FINDMAXIMUMSIZEDFROMSTART(data$[i_1, i_2]$, B)
    end for
    ▷ Select which block has the minimum average size per entry.
    $bestBlock$, $avgSize \leftarrow$MINIMUMAVERAGESIZE($potentialBlocks$)
    ▷ If the average entry takes more space then the full material, use that.
    ▷ Otherwise, finalize the block with the minimum average size per entry.
    if $avgSize < materialCost$ **then**
      $e \leftarrow$END($bestBlock$) ▷ End returns the position after the last entry
      FINALIZEPALETTEFORBLOCK($data[i_1, e)$
      $i_1 \leftarrow e$
    else
      $e \leftarrow$END($potentialBlocks[8]$) ▷ End returns the position after the last entry
      FINALIZEFULLMATERIAL($data[i_1, e]$)
      $i_1 \leftarrow e$
    end if
  end while
end function

Chapter 4

Implementation

We have implemented the methods described in Chapter 3 using C++ with OpenGL and GLSL. This section discusses implementation details, including how the DAG construction works, memory optimizations and rendering.

4.1 DAG/Octree

The elements involved in the compression, such as the octree structure, the quantization, the data compression, the voxelization algorithms, and the methods for efficient tree storage are implemented in a modular fashion. This is realized using the factory and adapter design patterns ([Gam+94]). Users can use a single string to specify the octree type, quantization parameters and data compression algorithm. A second string is used to specify what type of tree storage (naive, pointer sizes per level, pointer entropy encoding or virtual nodes) should be used.

4.1.1 Construction

The octree can be built from either a triangle mesh or a 3D grid (e.g. medical data).

Construction from a triangle mesh is implemented using *depth peeling* ([Eve01]), where the standard extension proposed by [HTG03] is applied. To do this, we first render the scene normally using an orthogonal projection. This gives us an image with triangle attributes (e.g., colors, normals) and a depth map. Using the depth map and the knowledge of the current view (direction, near- and far plane positions), we can reconstruct a voxel position in the grid. The next time the scene is rendered, a fragment (i.e., some pixel for some triangle) is discarded if its depth is smaller than or equal to the depth of the corresponding pixel in the last rendered depth map. This process is repeated until all fragments in a scene are discarded, and the resulting image is empty. The first three steps of the depth peeling are illustrated in Figure 4.1. This way, all

FIGURE 4.1: Example showing 3 steps of the depth peeling algorithm (from left to right), and a screenshot of the final 3D model.

overlapping triangles get rendered to an image, and consequently converted to voxel grid coordinates.

To make sure that planes parallel to the view direction are voxelized correctly, depth-peeling is executed in a forward fashion in the x, y and z direction (i.e., for view directions $\{[1,0,0],[0,1,0],[0,0,1]\}$. If some voxel appears in multiple view directions, its color is taken from the view in which the normal of triangle from which it originates has the smallest angle with the current view direction. This is done because triangles with normals that have a larger angle with the view direction, might sample the color from a lower resolution mipmap, leading to aliasing artifacts.

Construction from a 3D grid (e.g. medical data) is trivial. We use a transfer function to convert from data values to materials (color and opacity). When the opacity of a voxel is below some user-defined threshold, it is not added to the final tree. The octree is always built at the same resolution as the source grid. If the requested resolution is lower, the bottom levels of the tree are culled.

To allow for constructing high-resolution trees, for which the original octree does not fit into RAM, multiple steps are used. In each step, a subtree of the main SVO is constructed, compressed using the currently selected algorithm, stored on the hard drive, and cleared from RAM. This allows each subtree to be constructed with all system memory available. When all steps are completed, the subtrees are merged into the final octree.

The merging process is optimized so that if some subtree exists in the original and the merged tree, it is not copied to the merged tree. Instead, pointers to this equal subtree are updated when the parent nodes containing them are copied to the merged tree. This ensures that if the trees to be merged are optimal DAGs, the final tree is also an optimal DAG, improving performance and reducing memory consumption. Similar optimizations are applied to the data structures in Section 3.4, where equal palettes (for palette compression), blocks (for repeated-block compression) or subtrees (for bittree based compression) are not copied when trees are merged.

4.1.2 Memory storage

Even when using stepped construction, memory usage is an issue, especially during the final step where all subtrees are merged. We found that often, even 32GB of RAM is not enough to store the data structure. To this extent, we have implemented several optimizations that reduce the memory footprint of SVO/DAG nodes.

We found that nodes in a SVO on average have about 4 of the possible 8 children. Therefore, by using a dynamic array, we save the equivalent of storing 4 pointers each node, at the expense of storing a pointer to the dynamic array (as it cannot be part of the fixed-size object).

In order to use all available memory, we compile to a 64-bit architecture. However, we assume that there are never more than 2^{32} nodes at once in a DAG. Storing all nodes of a DAG consecutively in memory in a *node pool*, this assumption allows us to use 32-bit unsigned integers as child pointers, saving 4 bytes each pointer, for an average of 16 bytes each node. Whenever a node is no longer needed, for example during DAG compression, it is marked as unused in the node pool. After all nodes that are no longer needed are marked, the unmarked nodes are moved so that they appear consecutively from the start of the node pool, occupying the space earlier held by marked nodes. The node pool is then shrunk to the minimum size required to contain all unmarked nodes, freeing up resources. Each node stores its index, which is required to reconstruct the child pointers when the node pool is shuffled, sorted or shifted.

Nodes are allowed to insert new items into the node pool. Inserting a node requires the node pool to be resized, which could require a move operation on the entire node pool. If an object is moved while it is executing a method, it can no longer safely access its members, leading to errors. Furthermore, a very large empty block of memory is not always available, in which case a `bad_alloc` exception is thrown. To prevent this and make sure that adding nodes to the node pool never requires moving the node pool, we created a new type of container, the block vector. The block vector uses fixed-size blocks of memory that are created once and never moved. Whenever more memory is required, a new block is assigned. The block vector keeps track of all assigned blocks. Although this structure is inefficient for many small vectors, it works well for applications where a single very large vector, such as the node pool, is needed.

In conclusion, a node now takes an average of 32 bytes plus additional payload (e.g. colors, normals, other attributes):

- 0-32 bytes for child pointers (4 bytes per child, 0-8 children). Nodes have 4 children on average, leading to an average size of 16 bytes.
- 8 bytes for the pointer to the child pointer dynamic array. For high-resolution trees, we often need more than 4GB of memory for storing all child pointers, the maximum that can be indexed with 32 bits. Therefore, a 64 bit architecture and pointers of this size are required.
- 4 bytes for the original node index. This is required to restore node pointers when the node order is changed.
- 2 bytes to indicate which tree this node is a part of. Remember that multiple trees can be in memory at the same time (for example during merging).
- 1 byte for the childmask.
- 1 byte for the level on which the node resides.

4.2 Renderer

The rendering is implemented using a real-time raycasting algorithm similar to Laine and Karras [LK10]. A difference with their implementation is that beam optimization and contour checking are omitted, as these are not relevant to the goal of this thesis. We use a (triangle-mesh based) skybox to simulate the surrounding environment and atmosphere. We implemented the algorithm in GLSL, using 3D textures with 1 byte of information per texel to store the tree structure. The width and height of the textures are chosen to be a power of two, as this allows the use of simple bitwise operators to convert a 1D pointer to a 3D texture coordinate. Since some of the required textures are bigger than 4GB, we need to use pointers larger than 32 bits. This was implemented using GLSL extension `GL_NV_gpu_shader5`, which adds support for 64 bit unsigned integers. Since not all GPUs support 3D texture sizes of over 1024 (for all dimensions), we split the texture into blocks of size $1024 \times 1024 \times 1024$ (1GB), which are stored in a sampler array.

In order to allow a single shader to render all our compression methods, we use the `#define` and `#ifdef` keywords. When the shader is loaded, the correct defines are set for the current type of tree, making sure that only the appropriate code is executed.

The main lighting features of our renderer are:

- **Hard shadows**, which are rendered by casting a ray from the hit voxel to the light source. If this ray intersects geometry, the pixel is in shadow.
- **Diffuse lighting** for scenes with normals.

- **Ambient occlusion** in two variations:
  - Screen-Space Ambient Occlusion (SSAO), which is a relatively cheap post-processing effect. It works by sampling depth values around each pixel. These values are compared to the depth of the current pixel. Samples with a smaller depth (i.e. are closer to the camera), contribute to the occlusion of the pixel.
  - Ray-traced Ambient Occlusion, which is has fewer artifacts, but is more resource heavy. From the primary ray-voxel intersection point, a secondary ray is shot in a direction sampled from a stratified uniform random distribution along a hemisphere around the normal of the intersected voxel. The rays contribute to the occlusion if they hit another voxel before some set threshold time. Note that more efficient ambient occlusion approaches and optimization can be applied, but our implementation is purely for demonstration purposes.
- **Indirect illumination**, which works similarly to ray-traced ambient occlusion. The main difference being that the voxels hit by secondary rays contribute to the color of the voxel, instead of the occlusion.
- **Reflections**. These are calculated by shooting a reflection ray, where the angle of incidence with the normal is equal to the angle of the reflected ray with the normal. The (shaded) color of the voxel that the reflected ray hits is combined with the original voxel color based on the reflectivity of the voxel.

Chapter 5

Results

Our methods are primarily aimed at scenes containing non-solid geometry, which means that the inside of objects is generally empty. In this context, we obtain datasets by voxelizing existing triangle meshes, as explained in Section 4.1.1, which results in sparse voxel octrees. While our compression schemes are capable of handling any kind of spatially coherent voxel data, in practice, we evaluate our method using color and normal information, which are crucial for many realistic lighting techniques.

For all graphs shown in this chapter, the original data is available as tables in Appendix A.

5.1 Compression

In this section, we compare the data usage of our techniques against naive approaches and existing state-of-the-art techniques. We often report the memory usage per voxel, where the number of voxels is equal to the number of nodes in an SVO representing some scene at some specific resolution after our voxelization. Even when comparing to other techniques, for fairness, we use our obtained number of nodes in the SVO.

The reported compression ratios are calculated as

$$\text{Compression ratio} = \frac{\text{Compressed size}}{\text{Uncompressed size}}. \tag{5.1}$$

This means that a compression ratio smaller than 1 or 100% means that compression is achieved.

We compare our results to existing state-of-the-art techniques in Figure 5.1, and more detailed results are available in Table 5.1, for which we voxelize four different scenes at multiple resolutions. We consider the citadel scene, which locally contains detailed geometry; the city scene, that has a more uniform distribution of detail; the San Miguel scene, which contains highly detailed geometry (the tree and plant foliage); and the arena scene, which is obtained from real world photographs of the Parisian *Arène de Lutèce* using floating scale surface reconstruction [FG14]. This is a representative set of different navigable scenes, which is the target application of our methods.

The memory usage shown here is for color data. In the top figure, we have applied our scene-specific quantization (Section 3.3.1) to obtain 12-bit values (4096 colors). This means that for the standard SVO implementation we have an 8-bit childmask, a 32-bit pointer, and a 12-bit or 24-bit color value (depending on if quantization is enabled) for every node – note that the leaf nodes have no pointer. The pointerless SVO contains only a childmask and the color value per node (20 bits for quantized, 32 bits for full colors). ESVOs store the same information as a standard colored SVO, with additional

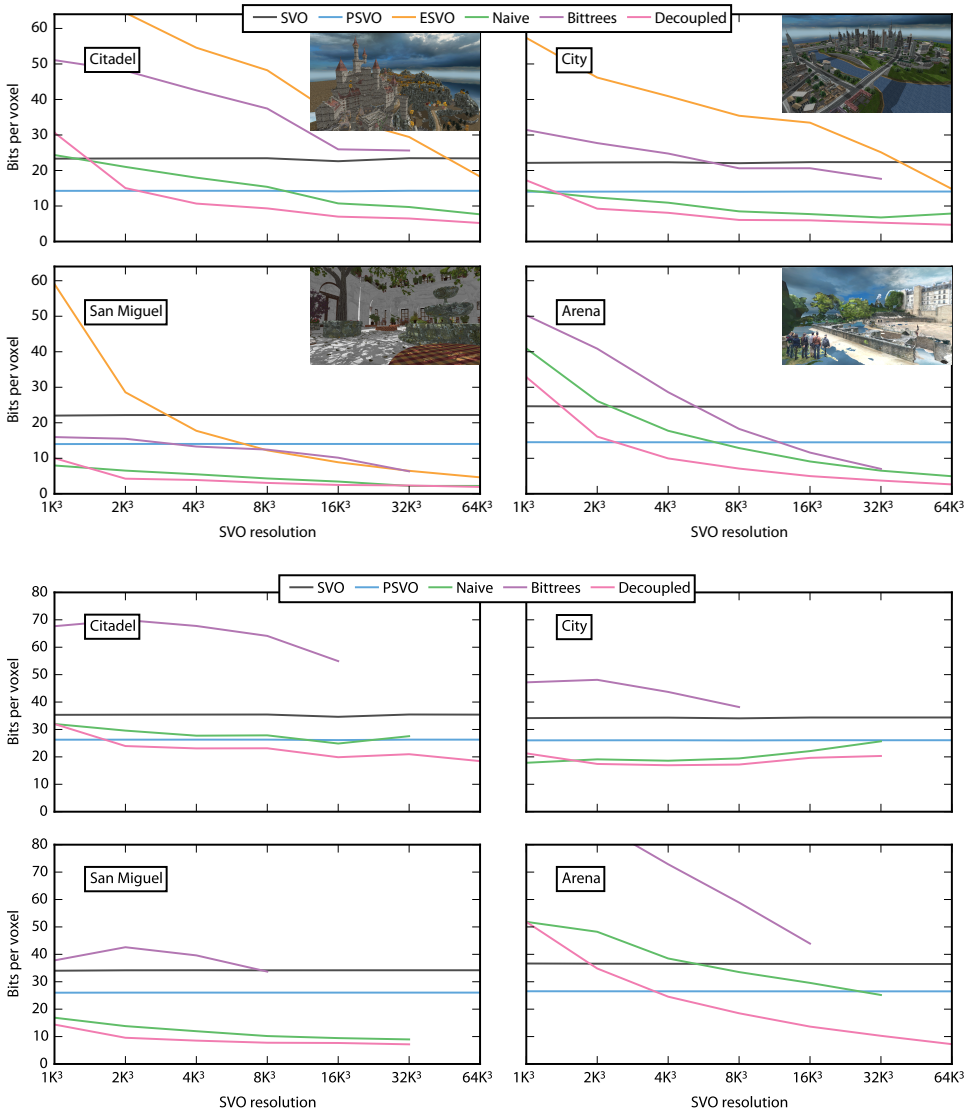


FIGURE 5.1: Memory usage comparison for several colored scenes at different resolutions. The top graph shows results for quantized colors (lossy compression), while the bottom graph shows results for lossless compression. We compare our approach to a standard colored SVO implementation, pointerless SVOs [SK06] (PSVO) and ESVOs [LK11]. Note that the ESVO implementation was unable to load the arena scene. The naive approach is a standard colored DAG (Section 3.1.1). Bittrees are implemented as explained in Section 3.1.2. The bittree data is incomplete due to the trees not fitting into memory. Decoupling corresponds with geometry-material decoupling of the tree (Section 3.1.3), using palette compression for the attribute data (Section 3.4.4).

Scene	Type	Number of voxels	4096 colors			Full colors		
			Size in MB		Bits/vox	Size in MB		Bits/vox
			Topology	Attributes		Topology	Attributes	
Citadel	SVO	4 760 302 085	2669.37	6809.67	2.92643	663.85	3371.94	4.43258
	PSVO	4 760 302 085	1295.14	6809.67	1.78529	322.04	3371.94	3.28652
	ESVO	4 760 302 085	8174.22		2.28507	-	-	-
	Naive	4 760 302 085	4335.00		0.95754	3859.00		3.44401
	Decoupled	4 760 302 085	348.00	2609.01	0.65136	123.00	2824.00	2.62193
City	SVO	10 487 130 645	5231.79	15001.96	2.79625	1305.97	7495.34	4.29587
	PSVO	10 487 130 645	2592.83	15001.96	1.75925	647.53	7495.34	3.25917
	ESVO	10 487 130 645	18505.80		1.85034	-	-	-
	Naive	10 487 130 645	9847.00		0.98485	8018.00		3.21039
	Decoupled	10 487 130 645	186.00	5703.01	0.58882	69.00	6280.00	2.54118
San Miguel	SVO	14 787 936 227	7219.71	21154.31	2.77403	1800.64	10560.78	4.27347
	PSVO	14 787 936 227	3593.51	21154.31	1.75481	896.59	10560.78	3.25469
	ESVO	14 787 936 227	10373.70		0.57961	-	-	-
	Naive	14 787 936 227	3896.00		0.27709	3931.0		1.12009
	Decoupled	14 787 936 227	316.00	3099.01	0.24215	104.00	3050.00	0.89596
Arena	SVO	3 263 192 560	2037.85	4668.03	3.05913	242.63	2332.72	4.56019
	PSVO	3 263 192 560	970.41	4668.03	1.81183	509.73	2332.72	3.31204
	Naive	3 263 192 560	1920.00		0.61793	2434.00		3.14568
	Decoupled	3 263 192 560	591.00	438.01	0.33066	195.00	804.00	1.28477

TABLE 5.1: A table corresponding with the high level data in Figure 5.1. Data for 4096 colors is at $64K^3$ resolution, data for full colors is at $32K^3$ resolution. Bittrees are omitted as high level data is missing.

contour data. However, as discussed in Section 2.1, to achieve results of a similar quality to regular colored SVOs, they can not cut off traversal as quickly as for geometry only.

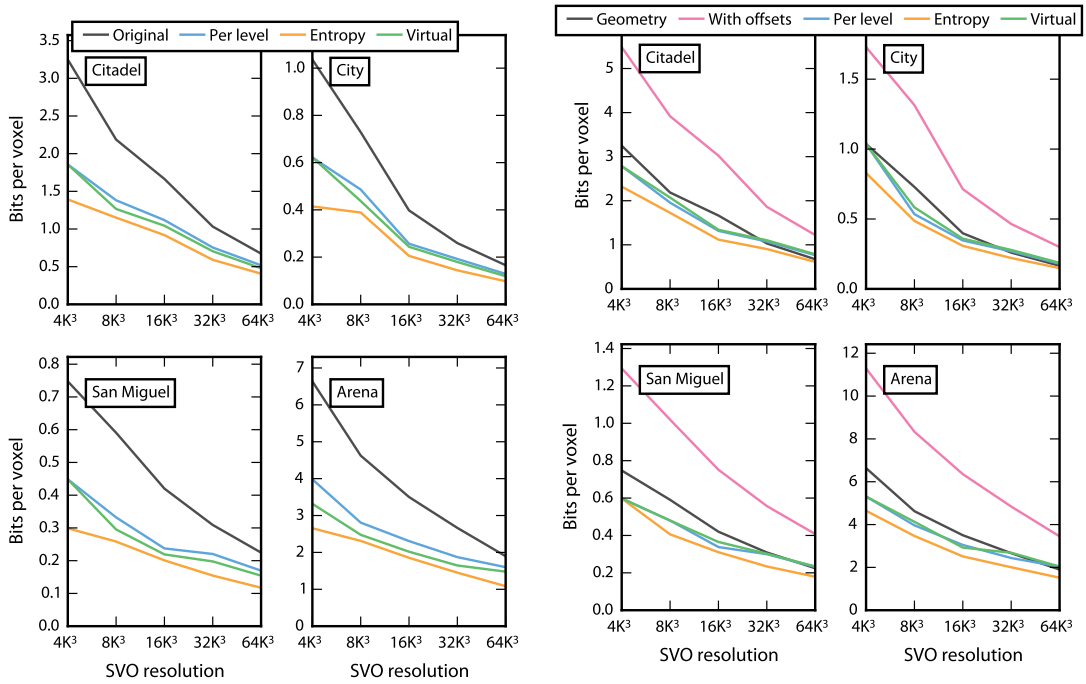
The naive method refers to the material DAG as proposed in Section 3.1.1. Bittrees are implemented as explained in Section 3.1.2. This means that, above the leaf nodes, only exactly matching subtrees are merged. Decoupling refers to a DAG in which the geometry and material information are decoupled, as proposed in Section 3.1.3. For compression of the node data table, we applied palette compression, as it works best in all tested scenes. For all our methods, we have used entropy encoding on the pointers.

Decoupling outperforms all other methods for the tested scenes, specifically in high resolutions. When comparing to a standard SVO, using 4096 quantized colors we obtain compression ratios of 23.7%, 21.1%, 8.8% and 11.4% for a $64K^3$ resolution in the citadel, city, San Miguel and arena scenes respectively. For full colors, we also obtain significant compression ratios of 51.0%, 59.1%, 21.0% and 28.2% for a $32K^3$ resolution. For this reason, we will evaluate this method in more detail.

5.1.1 Efficient tree storage

To evaluate the usefulness of our techniques to store the same tree more efficiently, as proposed in Section 3.2, we compare the memory consumption per voxel for a standard geometry DAG, and for a DAG that includes our offsets, enabling geometry-material decoupling. We compare the original approach by Kämpe et al., which uses 32 bits per pointer, to the techniques proposed by us; level precise pointer (and offset) sizes, using entropy encoding on the pointers, and using virtual nodes. Figure 5.2 shows these comparisons for the same four scenes. A table summarizing the compression ratio's at $64K^3$ resolution when comparing to the original DAG implementation is shown in Table 5.2.

We can conclude that our approach for topology encoding is efficient, reducing the memory requirements by 25-50% compared to the original DAG, depending on the resolution and method used. Moreover, despite the overhead that is caused by our geometry decoupling, using entropy encoding, we are able to encode the topology of voxelized scenes including offsets with a compression ratio of 91.1% (citadel), 89.9% (city),



(A) Memory usage for a geometry DAG

(B) Memory usage for a geometry DAG including offsets

FIGURE 5.2: Memory usage comparison for the pointer compression schemes, used for efficient tree storage, at different resolutions. Original refers to the naive method of using 32 bytes for offsets and pointers (Section 3.2.1). Per level refers to adapting the pointer and offset sizes to the level the node is in (Section 3.2.2). Entropy refers to using entropy encoding on the pointers (Section 3.2.3). Virtual refers to using virtual nodes to reduce the number of pointers in higher levels in the tree (Section 3.2.4). For both entropy encoding and virtual nodes, offset sizes are defined per level.

79.8% (San Miguel), and 80.0% (arena) when compared to the original DAG without offsets. In other words, our efficient tree storage can amortize the extra data required for geometry-material decoupling.

5.1.2 Data quantization

To empirically evaluate the quality of our quantization algorithms, we assert the mean and maximum errors. Since our color quantization is scene-dependent, we evaluate the introduced error on several voxel scenes with a $32K^3$ resolution. The results are presented in Table 5.3 in both RGB-space and as ΔE values. The ΔE values are calculated in accordance with the 1994 standard as formed by the CIE institute [Cie], using $k_L = 1$, $K_1 = 0.045$ and $K_2 = 0.015$ and D65 as the reference white. These are standard values for the graphics industry [KM10].

By definition, a ΔE value smaller than 1 cannot be perceived by a human observer. A value smaller than 2 corresponds to a minimal color difference. From the table, we see that using 16K colors leads to an average error that is not perceivable to human observers, and a maximum error that is hardly observable. Using 4096 colors does introduce perceivable errors in some areas, but it is still hardly noticeable to a human observer.

Scene	Original	Per level		Entropy		Virtual	
	Bits/Voxel	Bits	Compr.	Bits	Compr.	Bits	Compr.
Citadel	0.6732	0.5181	77.0%	0.4071	60.5%	0.4793	71.2%
City	0.1656	0.1288	77.8%	0.0976	58.9%	0.1192	72.0%
San Miguel	0.2246	0.1696	75.5%	0.1174	52.3%	0.1543	68.7%
Arena	1.8972	1.5990	84.2%	1.0848	57.2%	1.4781	77.9%
Citadel (+Offsets)	1.2217	0.7648	62.6%	0.6132	50.2%	0.7859	64.3%
City (+Offsets)	0.2995	0.1800	60.1%	0.1488	49.7%	0.1864	62.2%
San Miguel (+Offsets)	0.4077	0.2314	56.8%	0.1793	44.0%	0.2343	57.5%
Arena (+Offsets)	3.4497	2.0359	59.0%	1.5193	44.0%	2.0385	59.1%

TABLE 5.2: Table showing the average bits used per voxel when storing the topology in a DAG, and the compression ratio's comparing the original storage method proposed by Kämpe et al. to our proposed methods. All numbers are for the scenes at $64K^3$ resolution. The last four rows show the bits per voxel if offsets are included.

It should be noted, however, that banding might occur in some cases: although the maximum color difference between the original and quantized color is too small to be visible, the difference between different quantized colors is not. Therefore, if the original colors are smoothly varying in the scene, two noticeably different quantized colors might appear next to each other, which could cause banding artifacts. In practice, however, we found that using 4K colors is generally enough to prevent this.

The theoretical error for the normals is presented in **??**. To verify this result, we pick random samples uniformly on the surface of a unit sphere. These values are then stored as octahedral normal vector, and their stored value is compared to the original random sample. This yields the following mean/maximum errors (in degrees): 5.499/15.191 (10-bit), 2.711/7.471 (12-bit), 0.672/1.879 (16-bit), 0.041/0.117 (24-bit) and 0.007/0.044 (32-bit). Depending on the application, one might choose different bit depths for the normals. For diffuse lighting, 10-12 bits should be sufficient, especially for a scene with diverse colors. However, for smooth reflections or specularities, more detail is required.

5.1.3 Data compression

To evaluate our data compression techniques, a comparison is shown in Figure 5.3. We see that bittree-based compression is not beneficial: it never outperforms a trivial method such as tight packing. Repeated-block compression appears to be converging towards palette-based compression, but never outperforms it. Tight-packing is fast and has a constant performance, which is beneficial. However, this scheme does not exploit spatial coherence in any way. We conclude that palette based compression gives the best compression in all tested scenario's. Therefore, the remainder of this section will focus on the performance of the palette compression in different scenario's.

We evaluate the effectiveness of the palette compression technique when the voxels store normals rather than color data. In Figure 5.4, the memory usage when encoding normals is compared to that for colors for the citadel scene. We can conclude that even 16-bit normals require less memory than the colors, even though the colors need only 12 bits. This is due to the fact that often normals exhibit even more spatial coherence than colors. For the $16K^3$ resolution, we obtain memory usage for normals of 61.4% (16 bits), 42.6% (12 bits), 34.9% (10 bits) and 33.2% (8 bits) of the colors' memory usage (12 bits). We can conclude that it is relatively cheap to store normals.

Scene	# of colors	RGB error		ΔE	
		Mean	Max	Mean	Max
Citadel	256	(7.476, 6.229, 7.584)	(36, 28, 30)	3.5516	10.5425
	1024	(4.520, 3.726, 4.709)	(13, 11, 37)	2.1288	6.8747
	4096	(2.735, 2.230, 2.899)	(8, 7, 24)	1.2960	3.6978
	16384	(1.628, 1.307, 1.749)	(4, 4, 19)	0.7785	2.2813
City	256	(7.261, 5.800, 6.702)	(28, 20, 25)	3.6826	9.9246
	1024	(4.367, 3.476, 4.051)	(18, 15, 16)	2.2292	6.0236
	4096	(2.633, 2.080, 2.434)	(13, 9, 9)	1.3503	3.9748
	16384	(1.577, 1.231, 1.459)	(13, 2, 3)	0.8119	2.1581
San Miguel	256	(9.203, 7.840, 9.044)	(36, 27, 40)	4.4027	12.4536
	1024	(5.546, 4.641, 5.399)	(23, 18, 24)	2.6400	7.8288
	4096	(3.393, 2.805, 3.290)	(16, 12, 14)	1.6054	4.8918
	16384	(2.066, 1.694, 2.001)	(7, 7, 12)	0.9817	2.6535
Arena	256	(8.659, 6.873, 7.891)	(61, 14, 29)	4.1277	11.5306
	1024	(5.167, 4.091, 4.719)	(52, 10, 6)	2.4712	7.2108
	4096	(3.084, 2.414, 2.793)	(37, 2, 2)	1.4746	3.9748
	16384	(1.838, 1.421, 1.657)	(16, 3, 4)	0.8846	2.3797

TABLE 5.3: Differences between original and quantized values as a result of our color quantization, presented as the absolute errors in RGB-space and as ΔE values for a set of voxel scenes at $32K^3$ resolution.

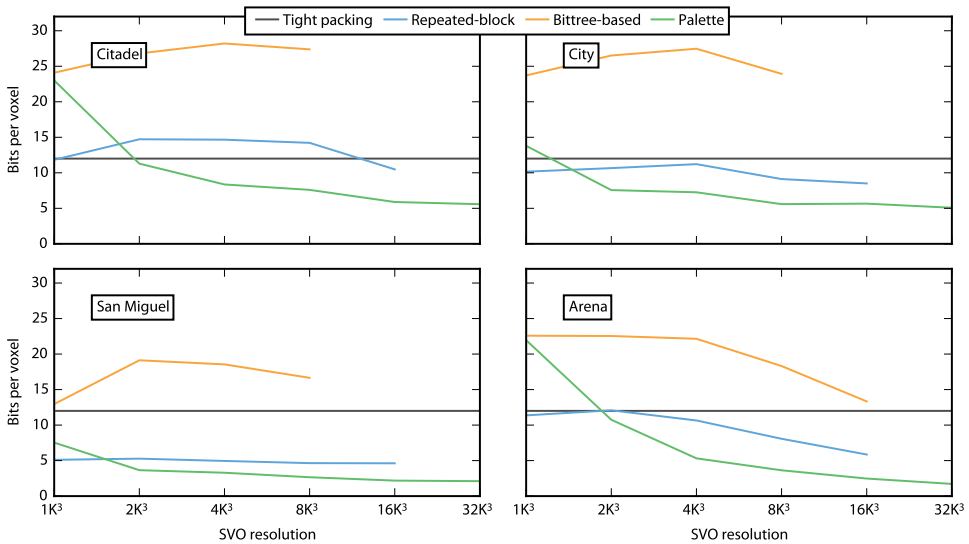


FIGURE 5.3: Comparison of the amount of data required to store the node data table when the techniques proposed in Section 3.4 are applied to our scenes with 4096 quantized colors. For repeated-block compression, blocks of size 8 are used, as empirical testing shows that this is optimal. Missing data is caused by faulty implementations for stepped construction. These problems were not resolved, as palette-based compression clearly outperforms the faulty methods.

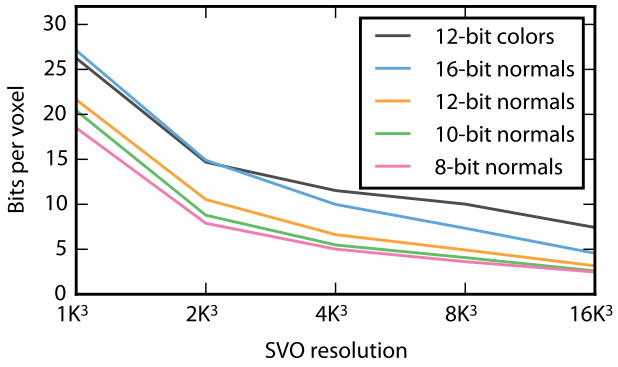


FIGURE 5.4: Memory usage comparison for normal and color encoding in the citadel scene at multiple resolutions. We consider 16-bit, 12-bit, 10-bit and 8-bit normals.

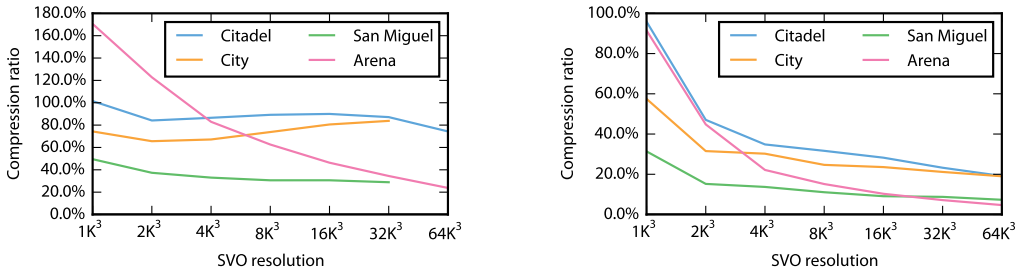


FIGURE 5.5: Compression ratios using our palette approach at different resolutions, using full colors (left) and 4096 quantized colors (right).

As it is interesting to see how our method behaves for larger voxel data, we also encode colors, normals, and reflectance information together for a special night-time version of the city scene. Using 4096 colors (12 bits), 10-bit normals, and 2 bits of reflectance information, we obtain a total memory footprint of 1492MB, compared to 1186MB for just encoding 4096 colors. This means that storing all this additional information generates a 25.8% overhead, while the initial voxel data is 250% larger. We can attribute this to the fact that a material consisting of similar colors often has the same reflectance value, and the normal and color values also correlate in many cases (e.g., a nearly uniformly colored wall).

The effectiveness of our palette compression scheme on color data for the same four scenes can be assessed in Figure 5.5. This data is based solely on the attribute data (i.e. the size of the node data table). Geometry is not included.

For quantized colors, the palette compression is always worthwhile, leading to compression ratios between 5% and 90%, depending on the scene and resolution. When using full colors, the palette compression successfully reduces the required size if the scene resolution is sufficiently high. We can conclude that for all scenes, using palettes is worthwhile. Using quantization to 12-bit colors, we achieve compression ratios of 19.2% (citadel), 19.0% (city), 7.3% (San Miguel), and 4.7% (arena) for 64K^3 resolutions. Without quantization, we get compression ratios of 83.8% (citadel), 83.8% (city), 28.9% (San Miguel) and 34.5% (arena) for 32K^3 resolutions. Remember that this does not yet include the compression achieved the DAG conversion of the geometry.

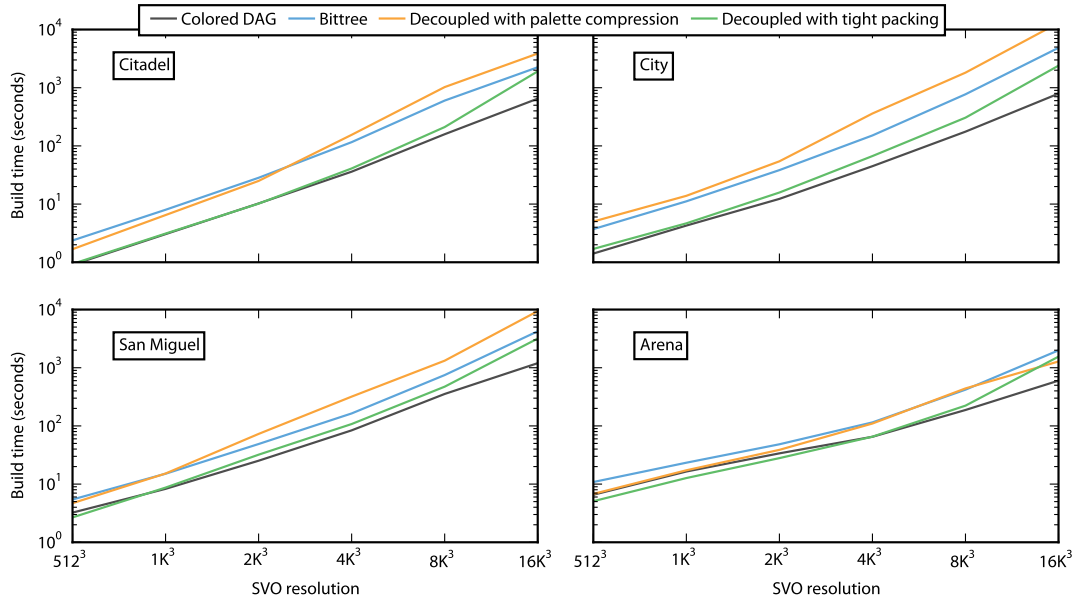


FIGURE 5.6: Complete construction times for the citadel, city, San Miguel and arena scenes at different resolutions and using different compression methods. All scenes were build using level-precise pointers, and color quantization to 4096 colors. The presented times are for complete construction, and thus include triangle scene loading and voxelization, DAG compression, color quantization, node data table compression

5.2 Construction times

Our techniques are aimed at offline construction. Therefore construction times are of lesser importance. With this in mind, we only optimized the code far enough so that high-resolution trees (e.g. $128K^3$) can be generated in a day on commodity hardware. Further optimization is possible, but was deemed unnecessary for this research. Build times up to $16K^3$ are presented in Figure 5.6.

As can be seen, buildtimes scale approximately linearly to the number of nodes in the tree (note the logarithmic axis). A discontinuity in the linearity of the construction times is visible going from $8K^3$ to $16K^3$, as this is where construction starts happening in steps, which requires storing and retrieving trees from disk. There is a lot of variation in the build times. In general, a standard colored DAG is quickest to build, while decoupling with palette compression takes the longest. A colored DAG representing the citadel scene at a resolution of $16K^3$ takes 10.8 minutes to construct, whereas constructing this scene using palette compression takes 64.5 minutes, or little over an hour.

Also notice that the difference between using tight packing and palette compression for the node data table is significant. At 14 levels for the city scene, for example, using palette compression took 5.43 times as long as using tight packing. Most of this time is required for the algorithm that finds maximum-sized blocks, which has a theoretical complexity of $O(N^2)$. Although this complexity is never reached in practice, it does illustrate the cost of this algorithm. The runtime is roughly proportional to the number and length of maximum sized blocks, which is significantly different per scene. The city scene, for example, has a large amount of water, which contains just a few shades of blue, allowing for many, large, maximum-sized blocks to be created with relatively small palettes; in fact, the current implementation finds a maximum sized block for every discontinuity in the node data table. This means that for every change in color in

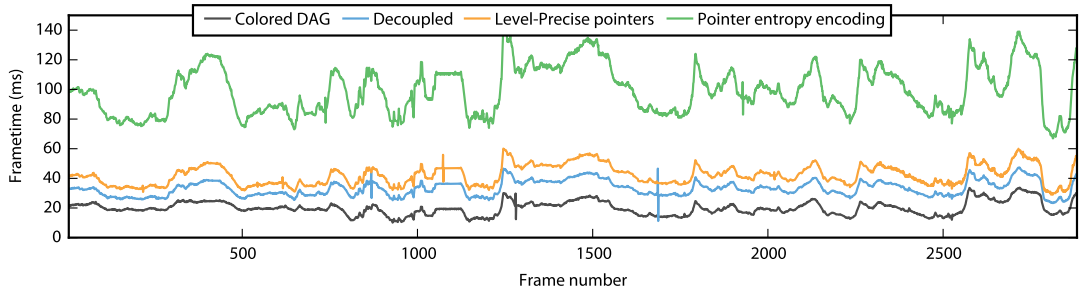


FIGURE 5.7: FPS while navigating through the citadel scene at a $32K^3$ resolution. Timings were obtained by raycasting in full HD.

the water, a new maximum-sized block containing the rest of the water is constructed. In contrast, the arena scene has a wider range of varying colors, which often appear in relative large blocks with the same color. This causes many blocks with 1 or 2 colors to be finalized early on, leaving much less data for analysis later on in the algorithm execution, and even making palette compression faster than tight packing.

5.3 Rendering performance

Since our main contributions lie within the realm of SVO compression, our rendering algorithm is not highly optimized; we simply cast rays from the camera and traverse the SVO using a standard stack-based approach to find the intersection with the first voxel that projects to an area smaller than a single pixel. Still, to demonstrate that our data structure is capable of real-time performance, we show the frame times for navigating through the citadel scene in full HD, at a $32K^3$ SVO resolution, in Figure 5.7. The timings were obtained using an NVIDIA Titan. We further assess the performance impact of our compression schemes by comparing the frame times when palette compression, pointer and offset sizes per level or pointer entropy encoding are enabled. We can conclude that pointer and offset sizes per level only have a small impact on the performance, while yielding significant compression rates. The entropy encoding on the other hand has a bigger influence; still, with a more optimized rendering algorithm it may be useful to reduce the memory footprint. Finally, our palettes also have little impact on the rendering performance, yet greatly improve the compression.

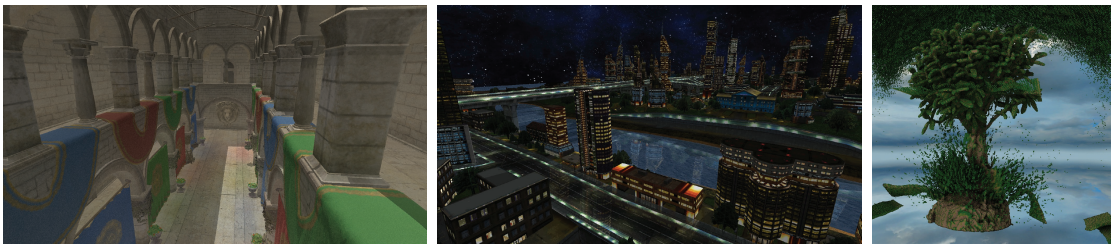


FIGURE 5.8: Several applications of our compressed SVO. From left to right: color bleeding in the CrySponza scene, at an SVO resolution of $4K^3$, using 16 samples per pixel, and secondary and tertiary ray tracing at a 512^3 resolution; encoding reflectance information in materials for the city scene, rendered at a resolution of $32K^3$; rendering of a dense volumetric dataset of a bonsai tree at a 512^3 resolution.

5.4 Applications

To demonstrate the usefulness of our approach, and using SVOs in general for storing 3D scenes, we showcase several applications. Like in the original DAG, we are able to obtain high-resolution hard shadows for the whole scene. For this, we shoot a secondary ray from a surface point hit by a primary ray, to the light source, resolving the visibility by traversing the SVO.

Now that we have colors and normals, however, we can look into more interesting applications. As showcased in the left image of Figure 5.8, we have implemented a simple approach to color bleeding through single-bounce global illumination. We shoot multiple secondary rays from a surface point hit by a primary ray and obtain the color of the first intersecting voxels. We then shoot a tertiary ray from this voxel towards the light source to determine if it is in shadow. The secondary rays are obtained by stratified sampling of the hemisphere, which means they are uniformly distributed, but contain a random offset. We further trace the secondary and tertiary rays at a lower SVO resolution, both to increase performance and decrease noise. While at full HD, we lose real-time performance, we still attain interactive rates for the setup shown in Figure 5.8. Note that our implementation is highly unoptimized and only serves to showcase an application of our SVO.

Besides colors and normals, we can also store more advanced material information in the SVO. As mentioned before, we have also included reflectance information for a night-time version of the city scene, results of which are shown in the center image of Figure 5.8. Besides the shadow rays, we now also shoot a secondary ray from the hit surface point in the reflected direction of the primary ray, and obtain the color of the first intersection voxel. In the case of Figure 5.8, we have added reflectance to the water, as well as to the roads, to make them look wet. As the performance overhead of shooting a single secondary ray is relatively small, we maintain real-time frame rates in full HD for the setup shown in Figure 5.8.

Since our method, like the DAG, exploits similarity as well as sparsity, we can potentially compress a greater variety of data. We aim at sparse navigable scenes, but as shown in the right image in Figure 5.8, our approach is also able to handle dense data. These datasets, however, are often available at a relatively low resolution (512^3 in this case) which decreases the compressibility. Still, for this data, we are able to obtain a compression rate of 68.4% without data quantization. Note that the density values in the data are first modulated by a transfer function, as is often done in medical visualization. As a result, our compression roughly corresponds to, for instance,

FIGURE 5.9: Encoding geometry at a different precision than colors. Left: SVO with 16 levels. Right: geometry is stored with 17 levels, of which only the first 16 are colored.

lossless compression of medical data as reported by Guthe et al. We have also applied our compression scheme, with a transfer function that removes the air and converts every opacity value to a unique color, to the christmas tree dataset presented in Guthe et al. [Gut+02], and obtain a compression ratio of about 10%.

As mentioned earlier, our geometry-material decoupling method allows us to use a different precision for the separate parts. This is useful for when the voxel data can be more coarsely encoded than the topology itself. Taking for instance the citadel scene of Figure 1.1, we can encode the geometry up to 17 SVO levels (voxel resolution $131,072^3$), while keeping the same resolution for the colors. This results in 19.2 billion nodes, 5.48 billion of which are colored, which corresponds to a total memory footprint of 3.75GB, which still allows for GPU storage. A visual comparison is shown in Figure 5.9.

Chapter 6

Conclusions

In this thesis, we present several techniques that allow for compression of SVOs. These techniques are all based on the DAG compression by Kämpe et al., yet allow for storing voxel attribute data. We apply our methods to color, normal, and reflectance information, but they are in no way limited to these attributes.

In short, we have three main compression schemes. The naive approach takes the standard DAG algorithm, but only allows merging subtrees when their attribute data and topology are identical. The bittree approach splits the material information in several parts (bits). For each of these bits we create an SVO. These SVOs are then merged into a single DAG, leading to compression. The decoupling approach stores the topology and attribute information separately. The topology is stored as a DAG in which offsets are added to all edges in the DAG. Summing these offsets from the root to a node allows us to find a unique index for each node in the original SVO, while maintaining the full DAG compression for the geometry. A separate node data table then stores the attribute information.

In addition to these schemes, we present several methods to reduce the data used to store pointers in the DAG. The most successful method is the use of entropy encoding on the pointers, achieving compression ratios of around 50% for all tested scenes. A downside of entropy encoding is that some additional operations are necessary to fetch a specific pointer, slowing down the rendering process. If, besides compression, performance is important, we recommend a simpler method, such as level-precise pointers. This effectively means that smaller pointers are used for levels that contain fewer nodes. We find that level-precise pointers lead to a compression ratio of about 75% compared to a standard DAG with 32-bit pointers, while only having a minor impact on rendering performance.

To allow for much better, but lossy compression, we introduce specialized quantization schemes for colors and normals. Our color quantization is scene specific, and is done in CIELAB-space. Using a perceptually (almost) uniform color space allows us to find quantized values with a minimal perceptual difference to their original values. This allows us to use 12 or 14-bit colors for most scenes without introducing disturbing quantization artefacts.

Arguably the most important element for our decoupling scheme is the method used for compression of the node data table. To this extent, we propose several algorithms, the most fruitful being palette compression. It uses variably-sized blocks, where each block only uses a small subset of all available materials. This allows the content of the block to be stored with very few bits per entry, leading to a significant compression.

All our methods provide compression compared to a standard SVO, and most techniques also improve over a pointerless SVO, especially for high resolution scenes. In general, using geometry-material decoupling with palette compression on a set of

quantized materials is most optimal, leading to compression ratios of between 8% and 25% for high resolution scenes. We obtain a minimal impact on rendering performance when using per-level pointer sizes.

Our compression schemes, for the first time, allow resolutions high enough for navigable voxel scenes, where voxels are not clearly visible, to be stored entirely in-core. Furthermore, rendering is possible in real-time, even allowing secondary rays to be cast for visibility queries. Complex lighting simulations, such as global illumination and reflections, can be performed in our DAG representation on commodity hardware at interactive framerates. The rendering performance of SVOs (and in extension DAGs) is, as opposed to triangle meshes, only minimally influenced by scene complexity. These properties all cater towards recent strides in the graphics industry for more detailed scenes and realistic lighting.

As the memory and speed of GPUs keep increasing, we believe that the use of voxel-based scene representations will increase. The techniques presented in this thesis will help accommodate with the memory requirements inherently present in these scenes. This, combined with novelty and effectiveness of our approaches should make them a valuable contribution to computer graphics research and the graphics industry.

6.0.1 Future work

For future work, it will be interesting to look at more advanced materials properties, like BRDFs encoded by spherical harmonics or transparency values. Furthermore, a lossy compression scheme that acts directly on the node data table could improve compression and reduce the need for prior quantization.

In addition, we believe that bittrees have not been fully explored. It might be possible to find a fast(er) approximation algorithm for merging subtrees when geometry is the only difference between them. It is also possible to merge subtrees on different levels, or even in a cyclic manner, losing the directionality of the DAG.

Finally, it might be interesting to explore real-time modifications to the compressed scene representation, allowing the user to add or remove voxels at will. This opens up possibilities for more applications, such as games with destructible environments or Minecraft-like gameplay.

On a more direct note, the performance can be improved for both the construction and rendering. For construction, a more efficient voxelization algorithm and caching mechanics can be used. In addition, memory assignment could be improved so that nodes do not have to move their child pointer array when a child is added or removed. For palette compression, it might be possible to detect regions with low and high variations, and finalize or discard those early on. This would reduce the amount of potential blocks that need to be analyzed. For rendering, one could incorporate beam optimization as proposed by Laine and Karras [LK10], experiment with aligning the nodes more optimally for caching purposes, use a short stack approach for SVO raycasting, exploit temporal coherency, and apply more general code optimizations.

In terms of applications, we think that current voxel-based games could benefit from a ray-casting approach. To this extent, it might be interesting to see the reactions of the Minecraft community if an "infinite" view distance renderer for their voxel worlds exists. This can be achieved by making the scene more sparse by culling invisible parts of the geometry (e.g. blocks that are not adjacent to see-through voxels), compressing this using our algorithms, and storing the entire world in-core.

Appendix A

Graph data tables

This appendix contains tables with the data as they are plotted in the graphs in Chapter 5.

A.1 Figure 5.1

	256^3	512^3	$1K^3$	$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$
Citadel	61 993	267 364	1 097 845	4 466 097	18 074 432	72 882 754	337 682 491	1 178 579 026	4 760 302 085
City	126 057	585 902	2 439 153	9 991 421	40 475 047	172 606 633	653 865 387	2 619 812 415	10 487 130 645
San Miguel	170 011	804 984	3 354 047	13 811 437	56 116 758	227 236 444	918 640 807	3 691 260 389	14 787 936 227
Arena	43 853	185 620	768 481	3 130 882	12 641 095	50 791 525	203 605 274	815 345 488	3 263 192 560

TABLE A.1: Table showing the number of voxels in our scenes at different resolutions.

A.1.1 4096 Quantized colors

	SVO	PSVO	ESVO	Naive	Bittrees	Decoupled
256^3	2.91242	1.78248		17.70729	6.54213	50.94149
512^3	2.90404	1.78081	11.18697	4.10574	5.98574	11.81167
$1K^3$	2.91681	1.78336	10.27110	3.04445	6.38433	3.83168
$2K^3$	2.92334	1.78467	8.05190	2.62667	6.01026	1.88104
$4K^3$	2.92872	1.78574	6.82250	2.24806	5.32228	1.33501
$8K^3$	2.93123	1.78625	6.02758	1.92428	4.68001	1.16553
$16K^3$	2.82673	1.76535	4.38263	1.34145	3.24461	0.87571
$32K^3$	2.93258	1.78652	3.67990	1.21443	3.20329	0.81052
$64K^3$	2.92643	1.78529	2.28507	0.95754		0.65136

TABLE A.2: Table showing the number of bytes per voxel used to store the citadel scene using 4096 quantized colors at different resolutions.

	SVO	PSVO	ESVO	Naive	Bittrees	Decoupled
256^3	2.76748	1.75350	9.74220	8.70819	3.70963	25.05229
512^3	2.73617	1.74723	9.08613	2.12524	3.18045	5.39001
$1K^3$	2.76780	1.75356	7.16826	1.80018	3.92757	2.15451
$2K^3$	2.78235	1.75647	5.77169	1.54798	3.46486	1.15565
$4K^3$	2.78897	1.75779	5.11713	1.36658	3.09647	1.01067
$8K^3$	2.75715	1.75143	4.42716	1.06160	2.57695	0.75944
$16K^3$	2.79467	1.75893	4.18338	0.96540	2.57999	0.74572
$32K^3$	2.79587	1.75917	3.14205	0.84653	2.20658	0.66243
$64K^3$	2.79625	1.75925	1.85034	0.98485		0.58882

TABLE A.3: Table showing the number of bytes per voxel used to store the city scene using 4096 quantized colors at different resolutions.

	SVO	PSVO	ESVO	Naive	Bittrees	Decoupled
256^3	2.65863	1.73173	9.56425	6.45681	2.43077	18.57536
512^3	2.71091	1.74218	9.06342	1.36366	1.90636	3.92308
$1K^3$	2.75232	1.75046	7.38110	0.99651	1.99660	1.25418
$2K^3$	2.77142	1.75428	3.57334	0.81615	1.93742	0.53234
$4K^3$	2.77420	1.75484	2.21780	0.68670	1.66380	0.48604
$8K^3$	2.77081	1.75416	1.53016	0.53989	1.55527	0.38306
$16K^3$	2.77262	1.75452	1.10972	0.43147	1.27142	0.31167
$32K^3$	2.77347	1.75469	0.80886	0.27782	0.79104	0.29232
$64K^3$	2.77403	1.75481	0.57961	0.27709		0.24215

TABLE A.4: Table showing the number of bytes per voxel used to store the San Miguel scene using 4096 quantized colors at different resolutions.

	SVO	PSVO	Naive	Bittrees	Decoupled
256^3	3.11711	1.82342	25.03199	7.79662	72.01368
512^3	3.09759	1.81952	6.70824	6.76482	17.01334
$1K^3$	3.08159	1.81632	5.11680	6.28396	4.10943
$2K^3$	3.07290	1.81458	3.26541	5.10847	2.01341
$4K^3$	3.06805	1.81361	2.21891	3.57748	1.24522
$8K^3$	3.06471	1.81294	1.61029	2.28451	0.88796
$16K^3$	3.06198	1.81240	1.13816	1.45057	0.62322
$32K^3$	3.06019	1.81204	0.81407	0.87598	0.46561
$64K^3$	3.05913	1.81183	0.61793		0.33066

TABLE A.5: Table showing the number of bytes per voxel used to store the arena scene using 4096 quantized colors at different resolutions.

A.1.2 Full colors

	SVO	PSVO	Naive	Bittrees	Decoupled
256^3	4.41242	3.28248	17.70729	7.40003	51.53614
512^3	4.40404	3.28081	4.10574	7.25186	11.94955
$1K^3$	4.41681	3.28336	3.99957	8.45897	3.99957
$2K^3$	4.42334	3.28467	3.69788	8.74189	2.99352
$4K^3$	4.42872	3.28574	3.46636	8.46970	2.88621
$8K^3$	4.43123	3.28625	3.48169	8.01566	2.89182
$16K^3$	4.32673	3.26535	3.10832	6.86636	2.48417
$32K^3$	4.43258	3.28652	3.44401		2.62193
$64K^3$	4.42643	3.28529			2.30672

TABLE A.6: Table showing the number of bytes per voxel used to store the citadel scene using the original colors (lossless compression) at different resolutions.

	SVO	PSVO	Naive	Bittrees	Decoupled
256^3	4.26748	3.25350	8.70819	4.92509	25.34473
512^3	4.23617	3.24723	3.91492	4.94244	5.70460
$1K^3$	4.26780	3.25356	2.23007	5.89901	2.65997
$2K^3$	4.28235	3.25647	2.38756	6.01353	2.17766
$4K^3$	4.28897	3.25779	2.32513	5.46139	2.11787
$8K^3$	4.25715	3.25143	2.42998	4.77169	2.15053
$16K^3$	4.29467	3.25893	2.76310		2.45520
$32K^3$	4.29587	3.25917	3.21039		2.54118
$64K^3$	4.29625	3.25925			

TABLE A.7: Table showing the number of bytes per voxel used to store the city scene using the original colors (lossless compression) at different resolutions.

	SVO	PSVO	Naive	Bittrees	Decoupled
256^3	4.15863	3.23173	6.45681	3.55514	18.79220
512^3	4.21091	3.24218	2.84945	3.94573	4.15205
$1K^3$	4.25232	3.25046	2.11025	4.72368	1.79762
$2K^3$	4.27142	3.25428	1.72720	5.32426	1.19575
$4K^3$	4.27420	3.25484	1.49485	4.95197	1.06508
$8K^3$	4.27081	3.25416	1.27359	4.21279	0.96904
$16K^3$	4.27262	3.25452	1.18025		0.95767
$32K^3$	4.27347	3.25469	1.12009		0.89596
$64K^3$	4.27403	3.25481			

TABLE A.8: Table showing the number of bytes per voxel used to store the San Miguel scene using the original colors (lossless compression) at different resolutions.

	SVO	PSVO	Naive	Bittrees	Decoupled
256^3	4.61711	3.32342	28.39450	14.23348	76.21682
512^3	4.59759	3.31952	15.53488	13.84110	21.18392
$1K^3$	4.58159	3.31632	6.48127	11.43029	6.48127
$2K^3$	4.57290	3.31458	6.02845	10.98383	4.35388
$4K^3$	4.56805	3.31361	4.81109	9.10832	3.06914
$8K^3$	4.56471	3.31294	4.19087	7.36167	2.31221
$16K^3$	4.56198	3.31240	3.69773	5.48944	1.70466
$32K^3$	4.56019	3.31204	3.14568		1.28477
$64K^3$	4.55913	3.31183			0.90327

TABLE A.9: Table showing the number of bytes per voxel used to store the arena scene using the original colors (lossless compression) at different resolutions.

A.2 Figure 5.2

A.2.1 Geometry DAG

	Original	Per level	Entropy	Virtual
256^3	135.31541	135.31541	135.31541	135.31541
512^3	31.37523	31.37523	31.37523	31.37523
$1K^3$	7.64098	7.64098	7.64098	7.64098
$2K^3$	5.63486	3.75657	1.87829	3.75657
$4K^3$	3.24880	1.85646	1.39234	1.85646
$8K^3$	2.18685	1.38117	1.15097	1.26607
$16K^3$	1.66439	1.11788	0.91914	1.04335
$32K^3$	1.03205	0.75446	0.59076	0.70464
$64K^3$	0.67316	0.51809	0.40707	0.47932

TABLE A.10: Table showing the number of bits required per voxel to store a geometry dag of the citadel scene at different resolutions.

	Original	Per level	Entropy	Virtual
256^3	66.54615	66.54615	66.54615	66.54615
512^3	14.31743	14.31743	14.31743	14.31743
$1K^3$	3.43915	3.43915	3.43915	3.43915
$2K^3$	1.67916	1.67916	0.83958	0.83958
$4K^3$	1.03627	0.62176	0.41451	0.62176
$8K^3$	0.72899	0.48600	0.38880	0.43740
$16K^3$	0.39771	0.25659	0.20527	0.24376
$32K^3$	0.25936	0.19212	0.14409	0.17931
$64K^3$	0.16558	0.12878	0.09759	0.11918

TABLE A.11: Table showing the number of bits required per voxel to store a geometry dag of the city scene at different resolutions.

	Original	Per level	Entropy	Virtual
256^3	49.34156	49.34156	49.34156	49.34156
512^3	10.42084	10.42084	10.42084	10.42084
$1K^3$	2.50104	2.50104	2.50104	2.50104
$2K^3$	1.21473	0.60737	0.60737	0.60737
$4K^3$	0.74742	0.44845	0.29897	0.44845
$8K^3$	0.59065	0.33224	0.25841	0.29533
$16K^3$	0.42005	0.23742	0.20089	0.21916
$32K^3$	0.30907	0.22044	0.15453	0.19771
$64K^3$	0.22464	0.16961	0.11742	0.15429

TABLE A.12: Table showing the number of bits required per voxel to store a geometry dag of the San Miguel scene at different resolutions.

	Original	Per level	Entropy	Virtual
256^3	191.28926	191.28926	191.28926	191.28926
512^3	45.19237	45.19237	45.19237	45.19237
$1K^3$	10.91583	10.91583	10.91583	10.91583
$2K^3$	10.71725	5.35862	5.35862	5.35862
$4K^3$	6.63598	3.98159	2.65439	3.31799
$8K^3$	4.62441	2.80768	2.31221	2.47736
$16K^3$	3.50203	2.30722	1.85402	2.01882
$32K^3$	2.66470	1.87249	1.45067	1.64615
$64K^3$	1.89716	1.59896	1.08482	1.47814

TABLE A.13: Table showing the number of bits required per voxel to store a geometry dag of the arena scene at different resolutions.

A.2.2 Topology and offsets

	Geometry	With offsets	Per level	Entropy	Virtual
256^3	135.31541	15.97342	135.31541	135.31541	135.31541
512^3	31.37523	12.55086	31.37523	31.37523	31.37523
$1K^3$	7.64098	9.91242	7.64098	7.64098	7.64098
$2K^3$	5.63486	7.59701	3.75657	3.75657	3.75657
$4K^3$	3.24880	5.48600	2.78469	2.32057	2.78469
$8K^3$	2.18685	3.91757	1.95665	1.72646	2.07175
$16K^3$	1.66439	3.02290	1.31661	1.11788	1.34145
$32K^3$	1.03205	1.86289	1.07475	0.90393	1.10322
$64K^3$	0.67316	1.22166	0.76480	0.61325	0.78594

TABLE A.14: Table showing the number of bits required per voxel to store a geometry dag of the citadel scene at different resolutions.

	Geometry	With offsets	Per level	Entropy	Virtual
256^3	66.54615	7.86539	66.54615	66.54615	66.54615
512^3	14.31743	5.55861	14.31743	14.31743	14.31743
$1K^3$	3.43915	4.01425	3.43915	3.43915	3.43915
$2K^3$	1.67916	2.69491	1.67916	1.67916	1.67916
$4K^3$	1.03627	1.72944	1.03627	0.82902	1.03627
$8K^3$	0.72899	1.31422	0.53460	0.48600	0.58319
$16K^3$	0.39771	0.71360	0.34639	0.30790	0.35922
$32K^3$	0.25936	0.46445	0.26897	0.22094	0.27857
$64K^3$	0.16558	0.29948	0.17998	0.14878	0.18638

TABLE A.15: Table showing the number of bits required per voxel to store a geometry dag of the city scene at different resolutions.

	Geometry	With offsets	Per level	Entropy	Virtual
$1K^3$	2.50104	1.66681	2.50104	2.50104	2.50104
$2K^3$	1.21473	1.55170	1.21473	0.60737	1.21473
$4K^3$	0.74742	1.29350	0.59794	0.59794	0.59794
$8K^3$	0.59065	1.01968	0.47990	0.40607	0.47990
$16K^3$	0.42005	0.75158	0.33787	0.31047	0.36526
$32K^3$	0.30907	0.55741	0.29998	0.23407	0.30225
$64K^3$	0.22464	0.40773	0.23144	0.17925	0.23428

TABLE A.16: Table showing the number of bits required per voxel to store a geometry dag of the San Miguel scene at different resolutions.

	Geometry	With offsets	Per level	Entropy	Virtual
$1K^3$	10.91583	18.15159	10.91583	10.91583	10.91583
$2K^3$	10.71725	15.04213	8.03793	5.35862	8.03793
$4K^3$	6.63598	11.29106	5.30879	4.64519	5.30879
$8K^3$	4.62441	8.33278	3.96378	3.46831	4.12894
$16K^3$	3.50203	6.35514	3.04883	2.51322	2.92522
$32K^3$	2.66470	4.84278	2.43835	2.00624	2.68527
$64K^3$	1.89716	3.44972	2.03597	1.51927	2.03855

TABLE A.17: Table showing the number of bits required per voxel to store a geometry dag of the arena scene at different resolutions.

A.3 Figure 5.3

	Tight packing	Repeated-block	Bittree-based	Palette
$1K^3$	12.00000	11.82913	24.09327	23.01247
$2K^3$	12.00000	14.72126	26.80287	11.29173
$4K^3$	12.00000	14.66422	28.20785	8.35950
$8K^3$	12.00000	14.21759	27.38106	7.59777
$16K^3$	12.00000	10.48365		5.88777
$32K^3$	12.00000			5.58025
$64K^3$	12.00000			4.59760

TABLE A.18: Table showing the memory usage in bits per voxel for the node data table using different compression techniques. The data is evaluated for the Citadel scene at several resolutions.

	Tight packing	Repeated-block	Bittree-based	Palette
$1K^3$	12.00000	10.15926	23.70504	13.79689
$2K^3$	12.00000	10.66054	26.51454	7.56607
$4K^3$	12.00000	11.21711	27.46825	7.25631
$8K^3$	12.00000	9.11949	23.93246	5.58952
$16K^3$	12.00000	8.50278		5.65785
$32K^3$	12.00000			5.07850
$64K^3$	12.00000			4.56181

TABLE A.19: Table showing the memory usage in bits per voxel for the node data table using different compression techniques. The data is evaluated for the City scene at several resolutions.

	Tight packing	Repeated-block	Bittree-based	Palette
$1K^3$	12.00000	5.10995	12.95816	7.53243
$2K^3$	12.00000	5.27423	19.12393	3.65132
$4K^3$	12.00000	4.95091	18.54717	3.29042
$8K^3$	12.00000	4.64368	16.65547	2.65837
$16K^3$	12.00000	4.62050		2.18287
$32K^3$	12.00000			2.10450
$64K^3$	12.00000			1.75795

TABLE A.20: Table showing the memory usage in bits per voxel for the node data table using different compression techniques. The data is evaluated for the San Miguel scene at several resolutions.

	Tight packing	Repeated-block	Bittree-based	Palette
$1K^3$	12.00000	11.36765	22.58227	21.95958
$2K^3$	12.00000	12.08067	22.54342	10.74864
$4K^3$	12.00000	10.65085	22.15036	5.31656
$8K^3$	12.00000	8.07248	18.31277	3.63540
$16K^3$	12.00000	5.85539	13.33009	2.47250
$32K^3$	12.00000			1.71865
$64K^3$	12.00000			1.12599

TABLE A.21: Table showing the memory usage in bits per voxel for the node data table using different compression techniques. The data is evaluated for the Arena scene at several resolutions.

A.4 Figure 5.4

	12-bit colors	16-bit normals	12-bit normals	10-bit normals	8-bit normals
256^3	34.12459	36.64129	34.06097	33.96555	33.43279
512^3	9.16420	9.98860	9.14888	8.80896	8.33500
$1K^3$	3.28137	3.38956	2.70470	2.55197	2.31342
$2K^3$	1.83535	1.86195	1.31654	1.09798	0.98603
$4K^3$	1.44097	1.24855	0.82850	0.68500	0.62742
$8K^3$	1.25253	0.91709	0.61584	0.50928	0.45273
$16K^3$	0.92815	0.57007	0.39496	0.32403	0.30844

TABLE A.22: Table showing the number of bytes required per voxel for storing the citadel scene. Comparisons are made between 12-bit colors, and several qunatization levels for normals.

A.5 Figure 5.5

A.5.1 Lossy compression

Res.	Citadel			City			San Miguel			Arena		
	Uncomp.	Comp.	Ratio	Uncomp.	Comp.	Ratio	Uncomp.	Comp.	Ratio	Uncomp.	Comp.	Ratio
256^3	0.18	2.01	10.92030	0.38	2.01	5.31114	0.50	2.01	4.02433	0.13	2.01	16.03418
512^3	0.76	2.01	2.62992	1.68	2.01	1.20011	2.30	2.01	0.87349	0.53	2.01	3.78810
$1K^3$	3.14	3.01	0.95885	6.98	4.01	0.57487	9.60	3.01	0.31385	2.20	2.01	0.91498
$2K^3$	12.78	6.01	0.47049	28.59	9.01	0.31525	39.51	6.01	0.15214	8.96	4.01	0.44786
$4K^3$	51.71	18.01	0.34831	115.80	35.01	0.30235	160.55	22.01	0.13710	36.17	8.01	0.22152
$8K^3$	208.52	66.01	0.31657	466.12	115.01	0.24674	650.13	72.01	0.11077	145.32	22.01	0.15148
$16K^3$	838.96	237.01	0.28251	1870.72	441.01	0.23574	2628.26	239.05	0.09095	582.52	60.01	0.10302
$32K^3$	3371.94	784.01	0.23251	7495.34	1586.05	0.21160	10560.77	926.05	0.08769	2332.72	167.05	0.07161
$64K^3$	13619.33	2609.01	0.19157	30003.92	5703.01	0.19008	42308.60	3099.01	0.07325	9336.07	438.01	0.04692

TABLE A.23: Table showing the compression ratio's using palette compression on several scenes and resolutions. Both the compressed and uncompressed sizes are reported (in MB). The compression ratio is also reported, and is defined as $Ratio = Comp/Uncomp$.

A.5.2 Lossless compression

Res.	Citadel			City			San Miguel			Arena		
	Uncomp.	Comp.	Ratio	Uncomp.	Comp.	Ratio	Uncomp.	Comp.	Ratio	Uncomp.	Comp.	Ratio
256^3	0.18	2.05	11.11116	0.38	2.05	5.40397	0.50	2.05	4.09467	0.13	2.19	17.43522
512^3	0.76	2.05	2.67589	1.68	2.19	1.30497	2.30	2.19	0.94982	0.53	2.75	5.17829
$1K^3$	3.14	3.19	1.01482	6.98	5.19	0.74336	9.60	4.75	0.49500	2.20	3.75	1.70560
$2K^3$	12.78	10.75	0.84132	28.59	18.75	0.65592	39.51	14.75	0.37328	8.96	11.00	1.22802
$4K^3$	51.71	44.75	0.86538	115.80	77.75	0.67142	160.55	53.00	0.33011	36.17	30.00	0.82950
$8K^3$	208.52	186.00	0.89200	466.12	344.00	0.73801	650.13	199.00	0.30609	145.32	91.00	0.62622
$16K^3$	838.96	755.00	0.89992	1870.72	1507.00	0.80557	2628.26	805.00	0.30629	582.52	270.00	0.46350
$32K^3$	3371.94	2824.00	0.83750	7495.34	6280.00	0.83785	10560.77	3050.00	0.28880	2332.72	804.00	0.34466
$64K^3$	13619.33	10124.00	0.74336							9336.07	2220.00	0.23779

TABLE A.24: Table showing the compression ratio's using palette compression on several scenes and resolutions. Both the compressed and uncompressed sizes are reported (in MB). The compression ratio is also reported, and is defined as $Ratio = Comp/Uncomp$.

A.6 Figure 5.6

	Colored DAG	Bittree	Decoupled, palette compression	Decoupled, tight packing
256^3	0.50200	0.73400	0.49400	
512^3	0.90400	2.36900	1.67800	0.93600
$1K^3$		7.98400	6.47700	3.11200
$2K^3$	10.25500	28.31900	24.99700	10.16800
$4K^3$	36.03800	115.99300	154.93400	40.94400
$8K^3$	158.73200	601.13100	1025.35000	210.74700
$16K^3$	649.09400	2247.11000	3867.31000	1911.68000

TABLE A.25: Table showing the buildtimes (in seconds) of the citadel scene for different types of trees.

	Colored DAG	Bittree	Decoupled, palette compression	Decoupled, tight packing
256^3	0.58100	1.36300	0.97600	
512^3	1.41100	3.71300	5.01500	1.69400
$1K^3$	4.27200	11.15000	13.86900	4.66700
$2K^3$	12.28400	38.35600	54.32500	15.79500
$4K^3$	44.79700	151.18000	360.35100	66.87000
$8K^3$	175.72400	771.09700	1815.36000	305.54800
$16K^3$	791.33700	4851.45000	13003.30000	2393.16000

TABLE A.26: Table showing the buildtimes (in seconds) of the city scene for different types of trees.

	Colored DAG	Bittree	Decoupled, palette compression	Decoupled, tight packing
256^3	2.03600		2.62500	
512^3	3.23500	5.43600	4.69900	2.67500
$1K^3$	8.28200	15.07100	15.20300	8.70900
$2K^3$	25.22000	48.85400	72.84500	32.02500
$4K^3$	83.86900	163.82800	318.78200	107.73700
$8K^3$	354.13900	747.99100	1321.49000	472.57300
$16K^3$	1198.12000	4234.69000	9391.90000	3173.02000

TABLE A.27: Table showing the buildtimes (in seconds) of the San Miguel scene for different types of trees.

	Colored DAG	Bittree	Decoupled, palette compression	Decoupled, tight packing
256^3	3.17800		2.24400	2.14800
512^3	6.56000	10.85600	6.78200	5.08200
$1K^3$	16.45400	23.31500	17.27100	12.70600
$2K^3$	33.75900	48.34000	38.77700	27.94600
$4K^3$	65.04700	114.98100	109.92200	65.29600
$8K^3$	187.77100	419.72700	441.08400	223.02900
$16K^3$	599.99300	1989.85000	1280.90000	1551.76000

TABLE A.28: Table showing the buildtimes (in seconds) of the arena scene for different types of trees.

Bibliography

[BR+14] M. Balsa Rodríguez et al. "State-of-the-Art in Compressed GPU-Based Direct Volume Rendering". In: *Computer Graphics Forum* 33.6 (2014), pp. 77–100. DOI: 10.1111/cgf.12280.

[Cie] *Commission Internationale de l'Eclairage (CIE)*. http://www.cie.co.at/. Accessed: 2015-12-08.

[Cig+14] Zina H. Cigolle et al. "A Survey of Efficient Representations for Independent Unit Vectors". In: *Journal of Computer Graphics Techniques* 3.2 (2014), pp. 1–30.

[Cra+09] Cyril Crassin et al. "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering". In: *Proc. of I3D*. 2009, pp. 15–22. DOI: 10.1145/1507149.1507152.

[EGG08] Fabio Marton Enrico Gobbetti and José Antonio Iglesias Guitián. "A Single-pass GPU Ray Casting Framework for Interactive Out-of-core Rendering of Massive Volumetric Datasets". In: *The Visual Computer* 24.7 (2008), pp. 797–806. DOI: 10.1007/s00371-008-0261-9.

[Eve01] Cass Everitt. *Interactive order-independent transparency*. Tech. rep. NVIDIA Corporation, 2001.

[FG14] Simon Fuhrmann and Michael Goesele. "Floating Scale Surface Reconstruction". In: *Trans. on Graphics* 33.4 (2014), p. 46. DOI: 10.1145/2601097.2601163.

[Gam+94] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[Gon85] Teofilo F Gonzalez. "Clustering to minimize the maximum intercluster distance". In: *Theoretical Computer Science* 38 (1985), pp. 293–306. DOI: 10.1016/0304-3975(85)90224-5.

[Gut+02] Stefan Guthe et al. "Interactive rendering of large volume data sets". In: *Proc. of VIS*. 2002, pp. 53–60. DOI: 10.1109/VISUAL.2002.1183757.

[HTG03] Bruno Heidelberger, Matthias Teschner, and Markus H Gross. "Real-Time Volumetric Intersections of Deforming Objects". In: *Proc. of VMV*. 2003, pp. 461–468.

[JT80] Chris L Jackins and Steven L Tanimoto. "Oct-trees and their use in representing three-dimensional objects". In: *Computer Graphics and Image Processing* 14.3 (1980), pp. 249–270. DOI: 10.1016/0146-664X(80)90055-6.

[KM10] Georg A Klein and Todd Meyrath. *Industrial color physics*. Vol. 154. Springer, 2010.

[KSA13] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. "High Resolution Sparse Voxel DAGs". In: *Trans. on Graphics* 32.4 (2013), p. 101. DOI: 10.1145/2461912.2462024.

[KSA15] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. "Fast, memory-efficient construction of voxelized shadows". In: *Proc. of I3D*. 2015, pp. 25–30. DOI: 10.1145/2699276.2699284.

[LH06] Sylvain Lefebvre and Hugues Hoppe. "Perfect spatial hashing". In: *Trans. on Graphics* 25.3 (2006), pp. 579–588. DOI: 10.1145/1141911.1141926.

[LH07] Sylvain Lefebvre and Hugues Hoppe. "Compressed random-access trees for spatially coherent data". In: *Proc. of EGSR*. 2007, pp. 339–349. DOI: 10.2312/EGWR/EGSR07/339-349.

[LK10] Samuli Laine and Tero Karras. *Efficient sparse voxel octrees – Analysis, Extensions, and Implementation*. Tech. rep. NVIDIA Corporation, 2010.

[LK11] Samuli Laine and Tero Karras. "Efficient sparse voxel octrees". In: *Trans. on Visualization and Computer Graphics* 17.8 (2011), pp. 1048–1059. DOI: 10.1109/TVCG.2010.240.

[Mea82] Donald Meagher. "Geometric modeling using octree encoding". In: *Computer Graphics and Image Processing* 19.2 (1982), pp. 129–147. DOI: 10.1016/0146-664X(82)90104-6.

[Mey+10] Quirin Meyer et al. "On Floating-Point Normal Vectors". In: *Computer Graphics Forum* 29.4 (2010), pp. 1405–1409. DOI: 10.1111/j.1467-8659.2010.01737.x.

[Nys+12] J. Nystad et al. "Adaptive Scalable Texture Compression". In: *Proc. of HPG*. 2012, pp. 105–114.

[SAM05] Jacob Ström and Tomas Akenine-Möller. "iPACKMAN: High-quality, Low-complexity Texture Compression for Mobile Phones". In: *Proc. of HWWS*. 2005, pp. 63–70. DOI: 10.1145/1071866.1071877.

[Sin+14] Erik Sintorn et al. "Compact Precomputed Voxelized Shadows". In: *Trans. on Graphics* 33.4 (2014), p. 150. DOI: 10.1145/2601097.2601221.

[SK06] Ruwen Schnabel and Reinhard Klein. "Octree-based Point-Cloud Compression". In: *Proc. of SPBG*. 2006, pp. 111–120. DOI: 10.2312/SPBG/SPBG06/111-120.

[SKU08] László Szirmay-Kalos and Tamás Umenhoffer. "Displacement Mapping on the GPU—State of the Art". In: *Computer graphics forum*. Vol. 27. 6. Wiley Online Library. 2008, pp. 1567–1592.

[Wil15] Brent Robert Williams. "Moxel DAGs: Connecting Material Information to High Resolution Sparse Voxel DAGs". In: (2015).

[Xia97] Zhigang Xiang. "Color image quantization by minimizing the maximum intercluster distance". In: *ACM Transactions on Graphics (TOG)* 16.3 (1997), pp. 260–276. DOI: 10.1145/256157.256159.