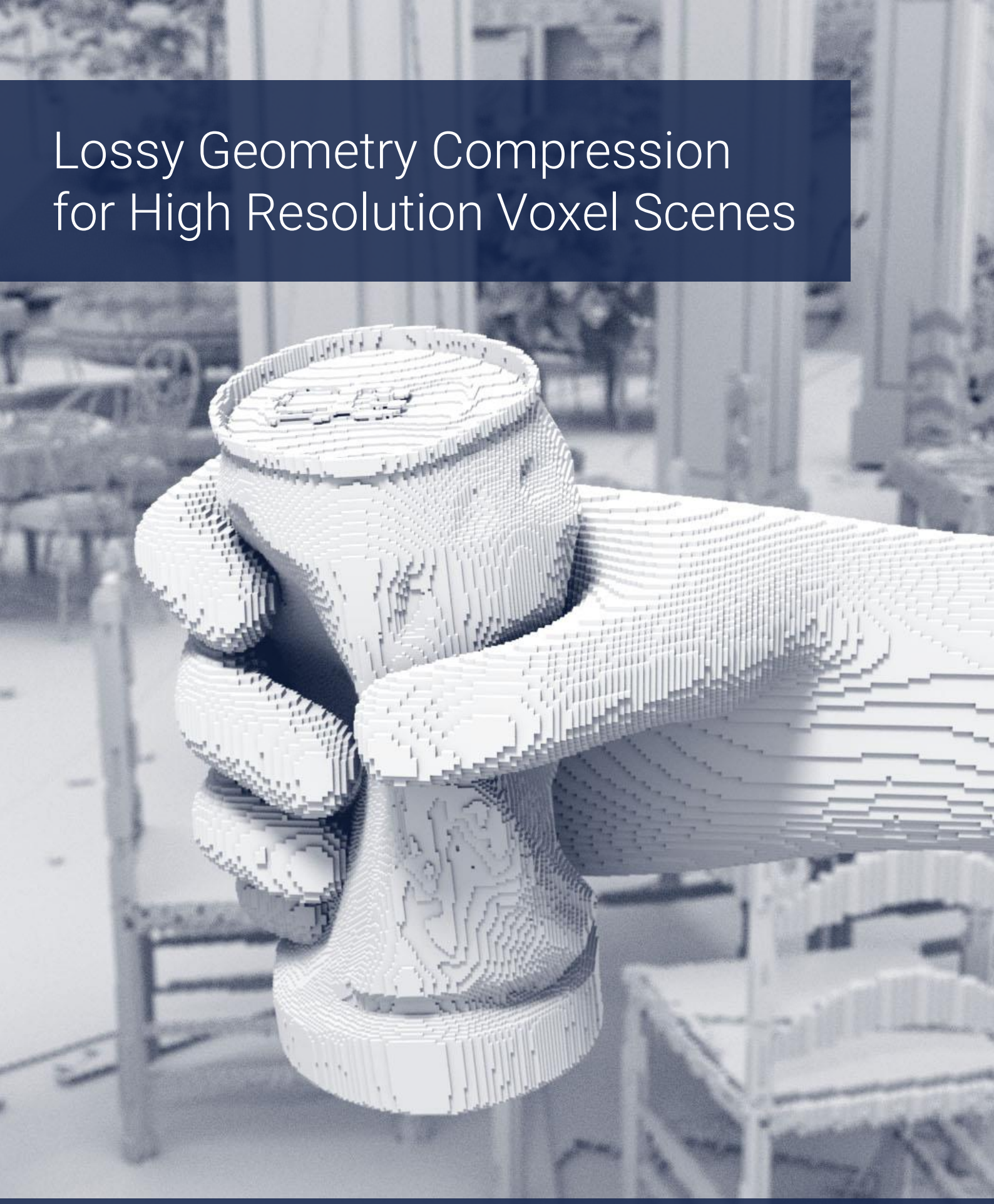
Lossy Geometry Compression for High Resolution Voxel Scenes



Master Thesis

Remi van der Laan

November 2019

TU Delft

Lossy Geometry Compression for High Resolution Voxel Scenes

by

Remi van der Laan

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday November 11, 2019 at 13:00 PM.

Research performed at:
Computer Graphics and Visualization
Department of Computer Science
Faculty of EEMCS
Delft University of Technology

Thesis committee:	Prof. dr. E. Eisemann,	TU Delft, Supervisor
	Assistant Prof. dr. R. Marroquim,	TU Delft
	Associate Prof. dr. P. S. Cesar Garcia,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Preface

Ever since learning how to programmatically paint pixels on a screen, I have gone on coding adventures to create interesting visualizations, which quickly grew into huge explorable worlds. There is something fascinating about exploring and interacting with a world where you understand every aspect of how it functions and is rendered to a screen. One of these aspects is space partitioning; how the regions of a world and the elements within it are organized. This was a topic that piqued my interest some time before having to choose a subject for my master's thesis. Therefore I am grateful for getting the opportunity to work on a topic relevant to my interests, and the experience I gained from spending over nine months of researching it.

First and foremost, my thanks goes out to my supervisor Elmar Eisemann for his guidance and the valuable and motivating discussions we have had. Additionally, I would like to thank Markus Billeter for providing me with useful suggestions and insights to some of the issues I encountered over the course of this research, and Alberto Jaspe Villanueva for kindly publicizing his well-crafted code that is used as the foundation of my implementation. Finally, thanks to Ruben Wiersma and my father for their rigorous proofreading, and the rest of my family, friends and fellow students for their support and the good times.

Remi van der Laan
Delft, November 2019

Abstract

Representing geometry data as voxels allows for a massive amount of detail that can be rendered in real-time. Storing this type of data as a directed acyclic graph (DAG) has recently led to immense improvements in memory consumption, which is one of the main limitations often associated with voxel-based approaches. We present a method for further decreasing the memory consumption of voxel data in a DAG through a form of lossy compression, meaning that the data is slightly altered from its original state. Our method clusters similar nodes in the graph together in a way that minimizes the geometric error that is introduced. The amount of compression that is applied can be influenced through a set of parameters that affect which nodes are included in the clustering process and the granularity of the clusters that are detected. The memory consumption of the sparse voxel DAG is reduced from 10% to up to 50%, depending on the characteristics of the scene, while introducing errors that are only visible when viewing the scene from up close. Our method is complementary to other state of the art compression methods for this data structure, and has a negligible effect on real-time rendering performance.

Contents

Acronyms

CSVDAG	Cross-level SVDAG	39
ESVDAG	SVDAG with pointer compression	8
ESVO	Efficient Sparse Voxel Octree ([LK11])	6
LSVDAG	Lossy SVDAG	13
SSVDAG	Symmetry-aware SVDAG ([VMG16])	9
SVDAG	Sparse Voxel Directed Acyclic Graph ([KSA13])	6
SVO	Sparse Voxel Octree	6

1

Introduction

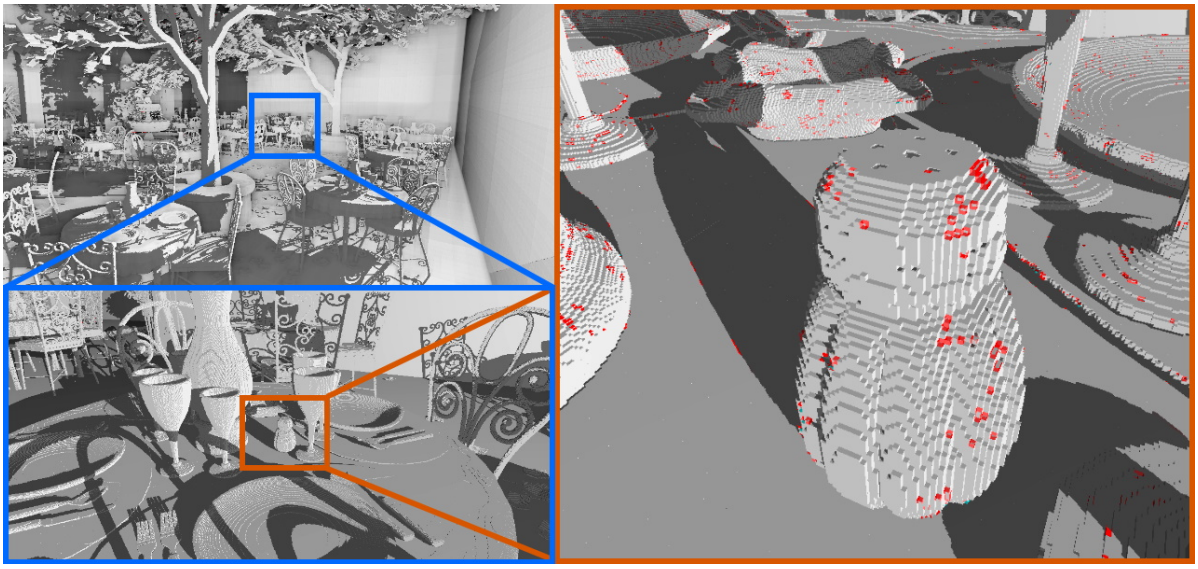


Figure 1.1: The San Miguel scene at a resolution of $64K^3$ compressed with our lossy compression technique. This scene contains a total of 12.4 billion non-empty voxels, stored in 939 MB of memory while a Sparse Voxel DAG requires 1228MB. The image on the right shows in red which voxels are different from those in the original scene, caused as a side effect from the lossy compression. The reduction in memory consumption is 23.61% while only 0.57% of all voxels is different from those in the original scene. The images are raycast in real-time with primary visibility rays and hard shadow rays.

1.1. Context

A highly sought-after goal in computer graphics is representing and rendering massively large scenes with a high amount detail. In the traditional rendering pipeline, where geometry is represented as triangles, the rendering performance is linked to the amount of triangles on the screen. This link between the amount of detail and the rendering performance severely limits the detail that can be achieved using this technique in large scenes. The real-time simulation of realistic lighting effects in these complex scenes, such as shadows, ambient occlusion and reflections, is not particularly well suited to be performed through the traditional rendering pipeline either. These effects are often approximated by ad-hoc solutions, or are completely omitted. Only recently have some of these effects been accurately reproduced in real-time using hardware accelerated ray-tracing [NVI18].

Voxelized representations offer an alternative approach to represent geometry, where the scene is represented as blocks, or voxels, on a 3D grid. The resolution of this grid controls the amount of detail of the geometry, which allows scenes to be incredibly detailed at high resolutions, at the cost of an

equally high memory consumption. However, the fact that most scenes are partially empty, or sparse, can be exploited to reduce the memory cost.

A commonly used spatial data structure that is relatively memory efficient is the Sparse Voxel Octree (SVO) [Mea82]. Several techniques based on the SVO have shown that it can be employed to render scenes with both a high amount of detail and with complex lighting effects in real time. It is an elegant representation compared to what is used in the traditional real-time rendering pipeline, where there is a separation between coarse geometry as triangles and high detail in textures. In an SVO, the attributes of voxels can be stored together with the geometry, in the same amount of detail as the geometry itself. Unfortunately, the memory consumption of an SVO does not scale up well beyond moderate resolutions.

The introduction of the Sparse Voxel Directed Acyclic Graph (SVDAG) sparked new interest into the voxel representation, as it could greatly compress high resolutions SVOs in a lossless manner [KSA13]. While initially limited to only representing geometry, several techniques have had success in assigning attributes to the voxels in the SVDAG [Wil15, DKB+16, DSKA17]. The SVDAG has also seen use in compressing 3D free viewpoint video [KRB+16] and the compression of precomputed shadows [SKOA14, KSDA16, SBE16a, SBE16b]. Compared to other compression techniques for 3D volumetric data, such as those described in an extensive survey by Balsa Rodríguez et al. [BRGIG+14], a major benefit of the SVDAG is that it has a negligible impact on rendering performance, as there is no need for decompression. The compression of the SVDAG is achieved by merging identical regions of space, which can be found all over most 3D scenes.

Some properties of the SVDAG have been exploited to further reduce its memory cost while minimally affecting traversal performance. The Symmetry-aware SVDAG (SSVDAG) [VMG16] losslessly compresses the SVDAG by merging nodes that are identical through a similarity transformation. Additionally, the authors of the SSVDAG and another research group [DKB+16] independently developed another lossless compression technique that reduces the size of pointers in the data structure. Our approach aims at exploiting additional properties, complementing the state-of-the-art techniques while also minimally affecting the traversal performance.

We propose a lossy geometry compression technique, that exploits the fact that a large majority of the nodes in the graph are only referenced once. In most scenes, a portion of these infrequently referenced nodes are very similar, which sometimes only differ by one or two voxels. In our technique, these similar nodes are clustered together and are replaced by a representative node. Our technique is configurable to control the amount of compression that is applied and can lower the memory cost from 10% to up to 50% to that the SVDAG, depending on the characteristics of the input dataset, with only a minimal loss in accuracy. In this work, we solely focus on geometry compression, not on the attributes of voxels, such as their color.

While a loss in quality is undesirable in some applications, in others the memory consumption may be more important. When the geometry in the scene will not be seen up close or when it contains noisy geometry, such as foliage or raw 3D scanned geometry, losing some detail in favor of a significantly lower memory consumption may be more favorable.

In addition to our main lossy compression method, we have performed research into two other compression methods. We propose a lossless compression method that merges nodes between all levels of the graph, instead of merging the nodes of each level separately. This method can reduce the amount of nodes in the SVDAG between 8% and 18% at a resolution of $64K^3$. Lastly, we experimented with a more efficient encoding of attributes in the bit-tree representation for assigning attributes to individual voxels, resulting in a consistent 1% to 10% improvement in memory consumption.

1.2. Contributions

Our main contributions are:

1. An algorithm for efficiently finding geometrically similar nodes in the SVDAG.

2. A novel variable lossy geometry compression method, compatible with existing state of the art compression methods.

1.3. Outline

The background information on which our method is based can be found in Chapter 2, starting with an explanation of the SVO and SVDAG data structures used for representing high resolution voxel scenes. Additionally, it provides an overview of lossy geometry compression for other data representations and a description of commonly used graph clustering algorithms. Our main method is described in Chapter 3, where we give insights into our decisions and explain the optimization of each of the steps of our approach. In Chapter 4, the architecture of the developed application and implementation details are described. Our technique is tested on a variety of datasets in Chapter 5, which is subsequently discussed in Chapter 6. We conclude this work in Chapter 7 with a brief summary and report the future work in Chapter 8.

Our research into two other compression techniques is described separately in Appendix A and B. While our main technique and these additional techniques can theoretically complement each other, they have been developed as stand-alone techniques and are therefore deemed to be better explained in their own self-contained chapters.

2

Background

There are three main research areas of interest to our proposed technique. The first and most relevant is the representation of high resolution voxel scenes, discussed in Section 2.1. This covers the evolution of a naive voxel grid to a greatly compressed high resolution Sparse Voxel DAG. Secondly, we provide an overview of lossy compression techniques used for other volumetric data representations in Section 2.2. The third topic is graph clustering, described in Section 2.3, which is one of the core components of our lossy compression technique.

2.1. High Resolution Voxel Scenes

A discrete 3D scene of resolution N along each axis contains N^3 cells, or voxels, meaning that the amount of voxels increases cubically when increasing the resolution N. Each voxel can either be full or empty, which can be represented with a single bit; 0 as empty or 1 as full. This indicates the presence of geometry in that voxel. Storing other kinds of information, such as its color, is beyond the scope of this thesis. Even when representing each voxel as a single bit of information, at a relatively low resolution of 1024^3 it would require over 100MB when naively storing the scene as a single long bit sequence.

The Sparse Voxel DAG (SVDAG) is a data structure which can efficiently encode high resolution voxel scenes at up to a resolution of $256K^3$ on consumer grade hardware with memory costs of a few gigabytes depending on the type of scene. A brief overview of how voxel scenes can be obtained is covered in Section 2.1.1. The SVDAG data structure is a modification of the Sparse Voxel Octree (SVO), which is be described in Section 2.1.2. Finally, the construction of the SVDAG itself and two effective lossless compression methods that can be applied to it are explained in Section 2.1.3.

2.1.1. Voxelization

The source of the voxels that inhabit a voxel scene can originate from many places. A source of many voxel scenes is the voxelization of polygonal triangle meshes. Triangle meshes are the industry standard representation in CAD programs for engineers and 3D modeling programs for artists, which provide tools that allow flexible workflows for the creation of highly detailed geometry and materials.

Voxelization is the process of discretizing a continuous surface or volume into a voxel representation. Schwarz and Seidel differentiate between two main types of voxelization, each for different contexts [SS10]. For volumetric data, the inside of geometry is just as relevant as the surfaces. Marking the voxels on the inside of solid objects as one is referred to as *solid voxelization*, which is useful for rendering objects with transparent or translucent materials. For non-transparent materials however, it would be more efficient to only store the outer surfaces of the geometry, which is understandably named *surface voxelization*. This is the type of voxelization that our technique is designed for.

2.1.2. Sparse Voxel Octree

Given a 3D voxel scene of resolution N^3, the simplest data structure would be to store every voxel individually in one long sequence. The value of any voxel can be directly looked up, as its index can

be computed purely based on its position in the grid. The problem with such a data structure is that a large portion of memory is wasted on empty parts of space.

Instead of storing all voxels of in a 3D scene individually in one long sequence, an octree recursively divides the volume up into 8 smaller cubical sub-volumes, or octants, up to a desired depth level. The depth level of the tree defines the resolution of the resulting voxel grid; for a level L, the resolution N equals $N = 2^L$ for every axis of the grid. Every node in the tree represents one of the sub-volumes of the volume, which gets smaller the deeper the level is at which the node resides. A simple example of an octree and the corresponding scene is visualized in Figure 2.1. We distinguish between three types of nodes; the root node is the first node of the tree, the leaves are those at the deepest level and the others are named the inner nodes.

(a) A naive voxel grid.

(b) An SVO constructed from the scene of Figure 2.1a. On the left of each level of the SVO, the reconstructed scene at that level is shown.

Figure 2.1: A voxel grid and a corresponding SVO, divided into three levels. Note that this example in 2D for the sake of clarity, causing this tree to classify as a quadtree. However, the same principles hold for an octree in 3D.

A Sparse Voxel Octree (SVO) is an octree that exploits the sparsity of a scene, usually meaning empty regions of space, that exist plentifully in the majority of 3D scenes. Nodes that represent an empty subvolume can be omitted from the tree, so they do not have to be stored in memory.

A benefit of the data structure being hierarchical for rendering is that it can be traversed with logarithmic time complexity, similar to a binary tree. Thus, the state of a voxel at a specific position, either empty or full, can be found relatively efficiently. Therefore, the SVO can be used to accelerate ray-casting; a technique that is typically applied to visualize a voxel grid, but can also be employed for producing complex lighting effects. Additionally, it provides implicit Level Of Detail (LOD), since the traversal can be terminated before reaching the leaves of the tree as a quality-performance trade-off, or when the resolution of the voxel grid is so high that a voxel appears smaller on the screen than a single pixel.

The Efficient Sparse Voxel Octree (ESVO) made great advancements in the compression of geometry, encoding of attributes and rendering of high resolution voxel scenes [LK11]. It is presented as an alternative to the triangle rasterization pipeline in terms of ray casting performance, while allowing tremendously greater geometric detail and unique shading information for every voxel. Besides just the presence of geometry in a voxel, the contours of the geometry are also encoded through a novel algorithm, which gives the illusion that the voxels are rendered as polygons instead of cubes. This also greatly reduces the amount of nodes needed to represent the scene, as subtrees can be culled early if the contour fits well to the original geometry. These contours are encoded for each voxel as two parallel planes, that match the orientation of the surface that intersects with the voxel.

2.1.3. Sparse Voxel Directed Acyclic Graph (SVDAG)

While Sparse Voxel Octrees offer many advantages, their main limitation is their massive memory consumption at high resolutions. Kämpe et al. prove that the memory requirements of the SVO data structure can be further reduced without losing information by turning the data structure into a DAG [KSA13]. A DAG can be used to model many types of information, but is especially useful for lossless compression in scenarios with data duplication. The Sparse Voxel Directed Acyclic Graph (SVDAG) not only encodes empty regions of space efficiently, but also identical regions, since it removes identical subtrees from the tree and points all references to a single representative subtree. This means that traversal trough a SVDAG is still very efficient, since it can be traversed in the same manner as an SVO. The reduction in amount of nodes is significant, but can vary greatly depending on the type

of scene and resolution; from a factor of 28 to 576 in the worst and best cases in the experiments performed by the authors.

A downside of SVDAGs is that the memory consumption of individual nodes is much larger than those of an SVO. In an SVO, all children of a node are laid out sequentially in memory, requiring only a single pointer per node in order to get the memory location of any child. Child nodes in the SVDAG are often children of multiple nodes, so they cannot always be laid out sequentially in memory. This means that every single child needs to have its own pointer per parent node. These pointers take up the majority of a node's size, usually 16 to 32 bits each. A node thus consists of an 8 bit children bitmask, and up to eight child pointers, resulting in up to 8 times as much memory consumption as nodes in the SVO. However, the memory improvements due to the removal of identical subtrees generally far outweigh the overhead of the larger sizes of nodes. On average, the SVDAG decreases the total memory consumption compared to the ESVO by a factor of 38 [KSA13].

Construction

An interesting property of an SVO is that individual voxels do not describe the geometry of the scene, it is the traversal path through the tree that is used to find the existence of geometry at a particular point in space. For this reason, merging identical subtrees of an SVO results in the same scene in the form of an SVDAG.

Reducing an SVO to an SVDAG poses an additional challenge. Since the algorithms consume temporary memory and we deal with large datasets, the whole SVO may not fit into memory. Therefore, the approach must be out-of-core; it should be able to compute partial results and merge them together afterwards into a final SVDAG. This out-of-core requirement is usually approached by building the tree up to a desired depth, reducing it to a DAG and merging it later with other reduced DAGs.

There are two main approaches on how the SVDAG can be constructed. The first is a relatively complex bottom-up approach, and the second, introduced more recently, is a somewhat simpler and more efficient top-down approach. These are illustrated in Figure 2.2.

Bottom-up An SVO can be converted into an SVDAG through a bottom-up approach, meaning that the construction takes place iteratively per level, starting at the leaf level. It is relatively complex to implement efficiently, since it requires sorting large sets of nodes.

The idea behind this approach is that the nodes at every level can be sorted, though which identical nodes can be easily identified. For the leaf level, the nodes can be sorted with respect to their bitmasks, which can be treated as integers. The inner nodes of the graph can be sorted based on their eight children pointers. For every set of duplicate nodes, one representative is chosen and the pointers are updated accordingly. This means that the algorithm is bounded by the amount of leaves in the SVO.

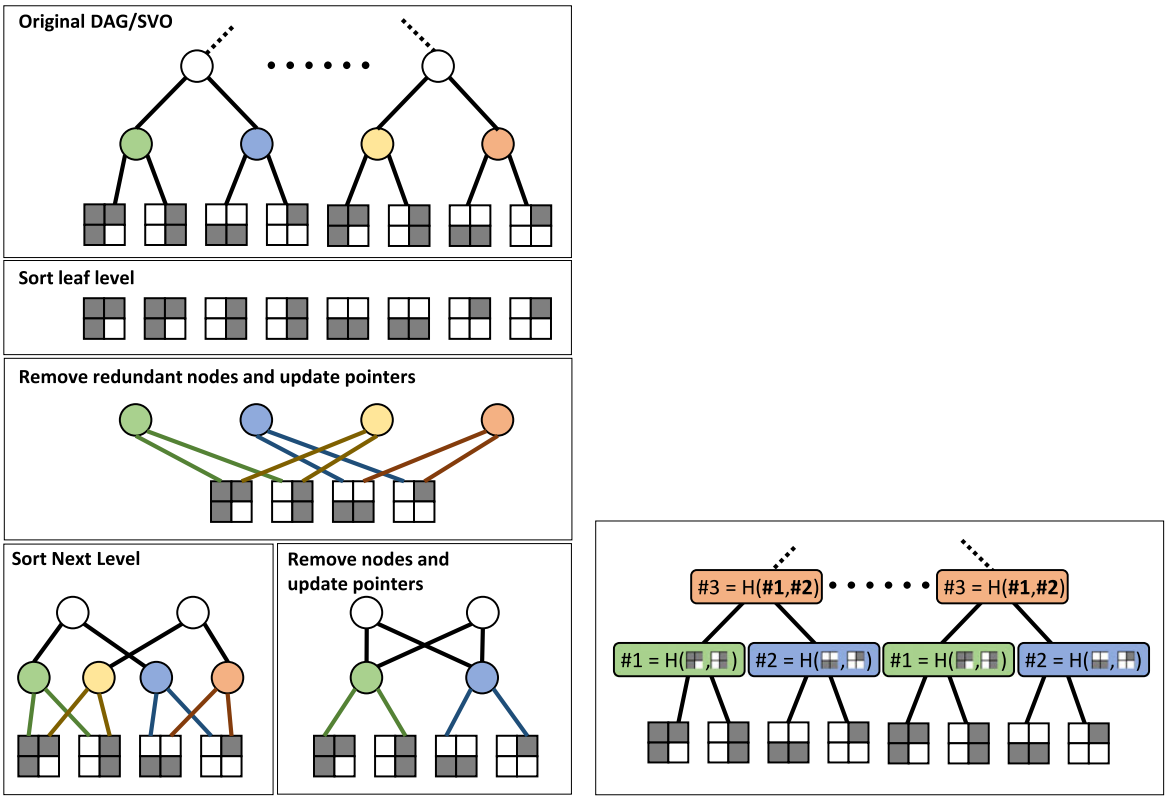
Top-down The top-down approach was introduced more recently by some of the same authors as the bottom-up approach [KSDA16]. It is based around the idea that identical subtrees can be immediately identified if they have identical hash values. These hash values are first computed for the whole tree, from bottom up, based on the bitmask for leaves and on the hash values of all children for internal nodes. Then, starting a top-down traversal at the root of the tree, identical subtrees can be immediately merged at any level if their hash values are equal.

This method generally performs about twice as fast as the bottom-up approach. The processing time here is proportional to largest level in DAG rather than the number of leaves in the SVO.

When there is little redundancy in the subtrees however, such as when merging multiple sub-DAGs together, this approach does not improve over the bottom-up approach. There is a theoretical possibility of hash collisions, but 64 bit hash values were sufficient in the authors' implementation for scenes of up to a resolution of $128K^3$ to avoid them.

2.1.4. SVDAG Compression

The SVDAG is an efficient representation due to the exploitation of sparsity in the scene and equality of subtrees. Still, it possesses some properties that can be exploited to compress this representation even more. There have been two major techniques that significantly reduce the memory needed to store the graph, both performed in a different domain. One accomplishes this by changing the memory representation of the nodes and pointers in the graph, while the other modifies the layout and construction of the graph by exploiting equality of subtrees through a similarity transformation.



(a) The steps of the bottom-up approach for reducing two SVOs or SVDAGs to a single SVDAG.

(b) The top-down hash-based approach for constructing an SVDAG, where nodes that have identical subtrees will have identical hashes.

Figure 2.2: The two types of construction methods for an SVDAG.
Source: [KSDA16]

Pointer compression

The size of pointers make up the majority of an internal node's size. Therefore, decreasing the size of these pointers would be a beneficial improvement to the overall size of the graph in memory. The SVDAG with pointer compression will be referred to as the ESVDAG.

The fact that certain subtrees are referenced far more frequently than others can be exploited to accomplish a reduction in the length of most pointers. Two research groups independently developed an entropy based encoding, by employing variable bit-rates for pointers based on how common a pointer is per level [VMG16, DKB+16]. To give an idea of how skewed these numbers are, in one arbitrary scene the authors mention that the 10% most referenced nodes are pointed to by 90% of all references.

The size of the most frequently used pointers is compressed by sorting the nodes in every level, so that the most frequently referenced nodes are positioned at the start. The pointers can be assigned a type which indicates the memory that it uses. The amount of memory that is saved lies roughly between 60% and 35% at resolutions from $2K^3$ up to $64K^3$, decreasing in overall effectiveness as the resolution grows.

A major drawback of this pointer compression is its performance impact on rendering, which decreases by around 15% in the first method and has an unspecified big influence in the second method as well. Villanueva et al. argue this decrease is caused in their method by the fetching of non-aligned data from GPU memory, since nodes are ordered purely based on their reference frequency, not on their position in space. The bitwise operations used to decode the pointers also introduce some additional costs.

Symmetry-aware compression

The Symmetry-aware Sparse Voxel DAG (SSVDAG) is a type of SVDAG that in addition to exploiting uniform and identical space also exploits parts of space that are identical to each other through a similarity transform [VMG16]. The similarity transformation chosen by the authors is symmetry along the main grid directions, but it could be extended to other types of similarity as well.

As with the original SVDAG, the authors propose a bottom-up approach to reduce an SVO into a minimal SSVDAG. However, instead of merging identical subtrees per level, nodes are merged based on a comparison which includes the symmetry transform. For a set of similar nodes, a representative node is chosen and all pointers to other similar nodes are pointed to that representative with the addition of a tag consisting of three bits. Each of these bits indicates whether a symmetry transformation should be applied for the X, Y and/or Z axis. A comparison of the graph layout of an SVO, SVDAG and SSVDAG is illustrated in Figure 2.3.

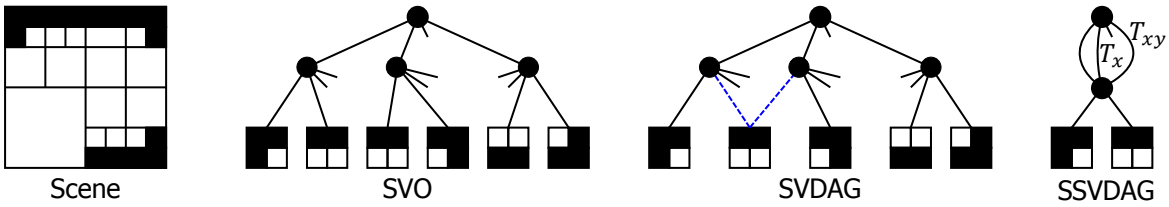


Figure 2.3: Overview of the layout of the SVO, SVDAG and the SSVDAG for a simple scene, shown as 2D quadtrees for clarity. **Source:** [VMG16]

The construction of the SSVDAG can start from either an SVDAG or SVO. The algorithm repeats the same steps for each level, starting at the bottom level that contains the leaf nodes. A bitmask indicating symmetrical invariance on each axis is computed for the nodes in this level, in order to avoid doing costly deep comparisons during the inner-node clustering stage when it is not necessary. Separate clustering methods are used for merging leaf nodes and for merging inner nodes.

For leaf nodes, similarity is defined as equality of their canonical representations. These canonical representations are precomputed for all possible 256 variations of a 2^3 grid of voxels.

For clustering inner nodes, the underlying subtrees must also be identical when a symmetry transformation is applied. First, all nodes of the level are sorted based on their child pointers, in order to position similar nodes nearby to each other. Then, for every node, a comparison is done to the representative nodes for all eight possible symmetry transformations. If no match is found, the node is added to the set of representative nodes.

The SSVDAG consistently outperforms the SVDAG in terms of memory usage. On average, it consumes around 79.6% of the memory compared to the SVDAG at a resolution of $64K^3$. When also employing the entropy based pointer compression described in Section 2.1.4, the total memory reduction compared to the SVDAG is about 52.4%.

The performance impact on rendering is negligible compared to the SVDAG when not using pointer compression, resulting in a decrease of $1\% - 2\%$ in the amount of rays cast per second.

2.2. Lossy geometry compression

Lossy compression can be applied to many types of data and can often reduce memory consumption in much greater amounts than types of lossless compression, at the cost of some loss in quality. How to quantify the loss, also referred to as the error or distortion, depends on the type of data that is processed. In this section we describe several lossy compression methods for two other types of volumetric data and explore how this loss can be measured. We limit ourselves to the compression of static geometry without attributes.

2.2.1. Triangle meshes

Geometry compression is often associated with 3D triangle meshes, as it is the most popular geometry representation. Spectral compression is one of the more commonly used approaches to obtain compressed 3D mesh representation, which is based on a spectral analysis of the mesh [KG00]. Its authors differentiate between a geometric and a visual loss metric. Commonly used geometric loss functions are the root-mean-square (RMS) error of the distances between vertices of two meshes, and

the Hausdorff distance which measures the maximum distance of any vertex to the surface of another mesh. As for measuring the visual loss, the authors of the spectral compression method develop a metric named the visual distance between meshes, where both the topology and geometry is taken into account. Though, they note that a single metric cannot be designed in such a way that it will be agreeable to the subjective visual systems of multiple observers. A more extensive overview of polygonal mesh compression can be found in the survey by Alliez et al. [AG05].

2.2.2. Point clouds

Point clouds are a fundamental representation of geometry, that can be found in many real-world applications such as in geographic information systems and autonomous driving. They share some similarities with voxel-based representations; both represent individual elements of a volume which means they both benefit from spatial partitioning, such as that of an octree. Schnabel et al. [SK06] base their lossless point cloud compression method based on an octree decomposition of space, where an arithmetic coding is used to outperform even lossy compression methods at the time. Another common approach used to achieve point cloud compression is the prediction of points based on their neighbours [GKIS05] or the prior edges in a spanning tree [MMG06]. The loss of point cloud compression techniques is conventionally measured as the difference in terms of the point-to-point or point-to-surface distance between the original and compressed point cloud. A recent survey by Tian et al [TOF+17] goes into more depth on geometric distortion metrics for point clouds.

2.3. Graph clustering

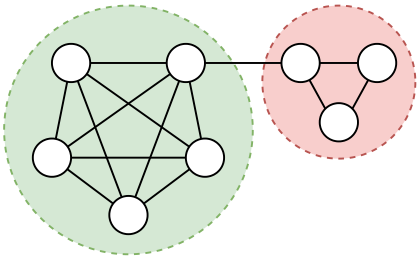


Figure 2.4: An example of an undirected graph where two detected clusters are surrounded by a circle of a different color.

The clustering of graphs, also known as community detection, is a research area that finds its use in many fields such as bioinformatics, sociology and data compression. We focus on clustering a particular type of graph; a weighted undirected graph. In this type of graph, pairs of nodes are connected to each other with a weighted edge that indicates how similar they are. An example of an undirected graph is shown in Figure 2.4, where the length of edges could be interpreted as their weight. There is a vast amount of clustering algorithms for this type of graph, therefore we will provide a brief overview of some of the more well-known ones. A more in-depth analysis of classic clustering algorithms and their applications can be found in a survey by Fortunato et al. [For10].

A widely used definition of a cluster is a group of nodes of a graph which are more strongly connected to each other than with other nodes in the graph. Graph clustering differs from graph partitioning, as for clustering the amount of clusters is decided by the algorithm and is not known beforehand.

The types of clustering algorithms suited for data compression have one or more parameters that affect the granularity of the clusters that are produced. Many fine-grained clusters result in higher quality and lower compression, while the opposite is the case for larger coarse-grained clusters.

2.3.1. Modularity

One of the most well-known clustering methods is modularity optimization, which optimizes a function that clusters should have more edges within the cluster than expected by chance [New04]. The modularity method was popularized through its use in the Louvain clustering algorithm [BGLL08]. This method maximizes the modularity by iteratively combining smaller clusters.

A major problem with modularity optimization algorithms is that they have trouble detecting small clusters in large networks, which is named the resolution limit [FB07]. Reichardt et al. addresses this problem by including a resolution parameter which makes it possible to influence the granularity at

which the clusters are detected [RB06]. More recently, Traag et al. prove that the Louvain algorithm can produce badly connected clusters even above the resolution limit, and introduce the Leiden clustering algorithm which guarantees to overcome this limitation [TWvE19].

2.3.2. Random walks

Detecting clusters through the simulation of random walks through a graph is a popular alternative to modularity optimization. The most well-known algorithm of this type is named Markov clustering [Don00]. Markov clustering detects clusters based on the assumption that there will be many edges between nodes within a cluster, and fewer edges to nodes outside of it. The clusters are detected by simulating a random walker through the graph, which should be more likely to stay within a single cluster than move out of it, based on the prior assumption.

Markov clustering is an iterative process that works on the so-called transfer matrix of the graph. The transfer matrix gives a probability that a random walker moves from any vertex to any other vertex. In each iteration, an expansion and inflation step are performed, which raises the values in the matrix to power and normalizes the values of each row respectively. After some amount of iterations, the values in the transfer matrix become either zero or one, turning it in an adjacency matrix, where the connected components are the clusters of the original graph. An inflation parameter can be used to decide the granularity of the clusters it produces.

The Markov clustering algorithm is well suited for graphs where the desired clusters are not too large. Additionally, graphs should not be so sparse that the degree of most nodes is close to one, or so dense that the degree of nodes is in the same range as the total amount of nodes in the graph.

3

Method

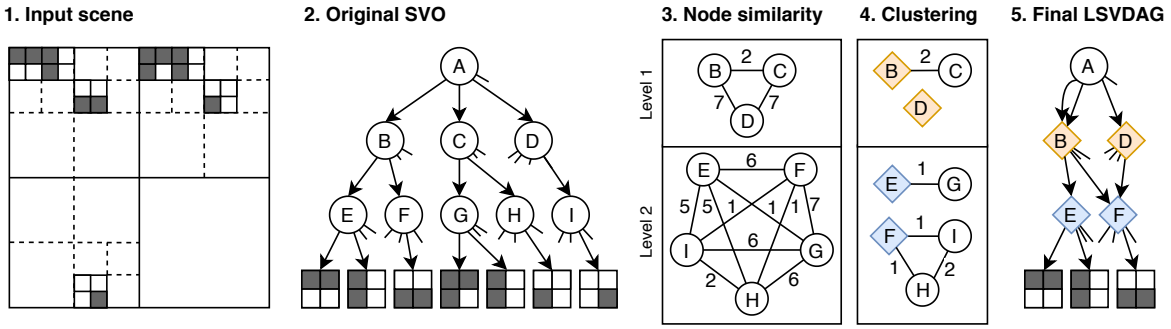


Figure 3.1: An overview of our lossy compression technique on a quadtree. 1. The scene used as input. 2. The Sparse Voxel Octree constructed from the input scene. 3. A weighted undirected graph of similar nodes is created per level, where the weights are measured as the difference in amount of voxels between them in the deepest level of the graph. 4. Clusters of similar nodes are identified. The cluster representatives are shown as diamond shapes, which have the least difference to all nodes in the cluster. 5. The nodes in each cluster are replaced by the cluster representatives in the SVO, resulting in our Lossy SVDAG.

Our method is a type of lossy compression that can be applied on an SVDAG, which we name the Lossy SVDAG (LSVDAG). Its aim is to produce a more compact representation of its input, while minimizing the geometric error that is introduced and minimizing the effect on traversal performance for rendering. We first provide an overview of our method in Section 3.1, followed by a definition of what it means for two nodes in the graph to be similar in Section 3.2. Then, we describe each of the steps of our method in more detail and how they are optimized. In Section 3.3 we discuss the methods for minimizing the amount of compression artifacts. Section 3.4 describes how we optimize the identification of similar nodes. How these similar nodes are clustered together is covered in Section 3.5. Finally, the parameters of our compression technique are described in Section 3.6, which can be tuned to influence the amount of compression and which nodes are affected by it.

3.1. Overview

The goal of our approach is to cluster similar nodes together, so that the total amount of nodes can be reduced. The simplest case where this can be done is at the leaf level. The leaf nodes can be directly compared based on the eight voxels that they represent. Then, groups of two or more of these nodes can be clustered together and replaced by a single representative node in the graph. However, the leaf nodes are the most common type of node in the scene, so clustering the leaf nodes together will cause huge differences to the voxels represented by all nodes further up into graph. Instead, it would be preferable to cluster uncommon nodes together so that the difference that is made on the amount voxels in the scene is minimized.

The inner nodes of the graph, those between the root and the leaf level, are more suitable for clustering, since many of these nodes are uncommon. Though, the similarity between these nodes

is harder to determine, as the difference between them needs to be measured based on the voxels at the deepest level of the graph. Therefore, the first step is to find which nodes are similar to each other, and the exact similarity between them. After similar nodes have been identified, the similarity between nodes can be modeled as a graph. The second step is to process this graph of the similarity between nodes with a clustering algorithm, which detects clusters of nodes of which it thinks are the most similar to each other. Then, in the last step, all nodes of a cluster in the SVDAG are replaced with their cluster representative; the node in the cluster with the highest similarity to all other nodes in the cluster. These steps are illustrated in Figure 3.1.

3.2. Node similarity

We describe the similarity between two nodes as a function of the difference between them in the voxels found at the deepest level of the graph of the subtree that they represent. Whether two nodes are considered to be similar is when this difference is below a specified threshold. This threshold can be defined in many ways, and can have a huge impact on the computation time and compression performance. An additional challenge for defining this threshold is that the measure of similarity between nodes needs to appropriate at every level in the graph.

We have decided to define the threshold for two nodes to be considered similar as a quadratic function of the depth d of the subtrees of a level. Thus, the maximum allowed difference M is computed as $M = d^2$. For example, at the level above the leaves, the subtrees have a depth of two, so the maximum allowed difference between two subtrees is 4 voxels, whereas the maximum amount of possible voxels at this depth is 64. We have opted for a quadratic function, as we aim to compress 2D surfaces in 3D space, which generally increase in the amount of voxels in a quadratic manner. Therefore, we believe it is the an appropriate threshold at every level of the graph. We have also experimented with a linear and a cubic function of the depth, which respectively cluster together subtrees too lightly or too aggressively as the depth of the subtrees per level increases. To control the amount of compression that is applied, we introduce a parameter p that linearly scales the threshold M, turning the equation for M into $M = d^2 \cdot p$. By default, p is set to 1. Lowering this parameter will decrease the compression ratio, error rate and computation time and increasing it vice versa.

Based on only this prior definition, a problem arises for nodes with an overall low amount of voxels in the leaves of their subtrees. With a threshold solely defined by the depth of subtrees in a level, a subtree with a single voxel would be considered similar to another subtree with a single voxel at a completely different location. For this reason, we introduce a second condition when comparing two subtrees, that enforces that each pair of leaf nodes at the same location of the two subtrees may not differ by more than four voxels; half the maximum amount of voxels per leaf node.

3.3. Minimizing compression artifacts

3.3.1. Exploiting reference counts

Our method relies on the property of the SVDAG that a large majority of its nodes is only referenced once. This skewed distribution of reference counts can be exploited, similarly to how the pointer compression technique described in Section 2.1.4 exploits the property most references point to a small set of nodes per level. In our case, we can apply clustering only on infrequently referenced nodes to minimize the impact it has on the amount of voxels in the scene.

The amount of times a node is referenced in the graph is not representative of how often the node appears in the scene. For instance, if a leaf node is referenced once, but one of its parents or grandparents is referenced twice, the leaf node will appear in the scene at least twice as well. Therefore we base our reference counts on what we name the *effective reference count*, which is how often a node would appear in the scene, and is identical to how often it would appear in the SVO that the SVDAG is based on. We use this metric since clustering nodes together based on their direct reference count could have a large effect on the amount of voxels that is affected if one of their parents is referenced many times. The distribution of effective reference counts for an arbitrary scene is shown in Figure 3.2. It shows that compared to the direct reference counts, the distribution of effective reference counts changes very little for nodes that are infrequently referenced, meaning that most nodes with a direct reference count of one appear in the scene only once.

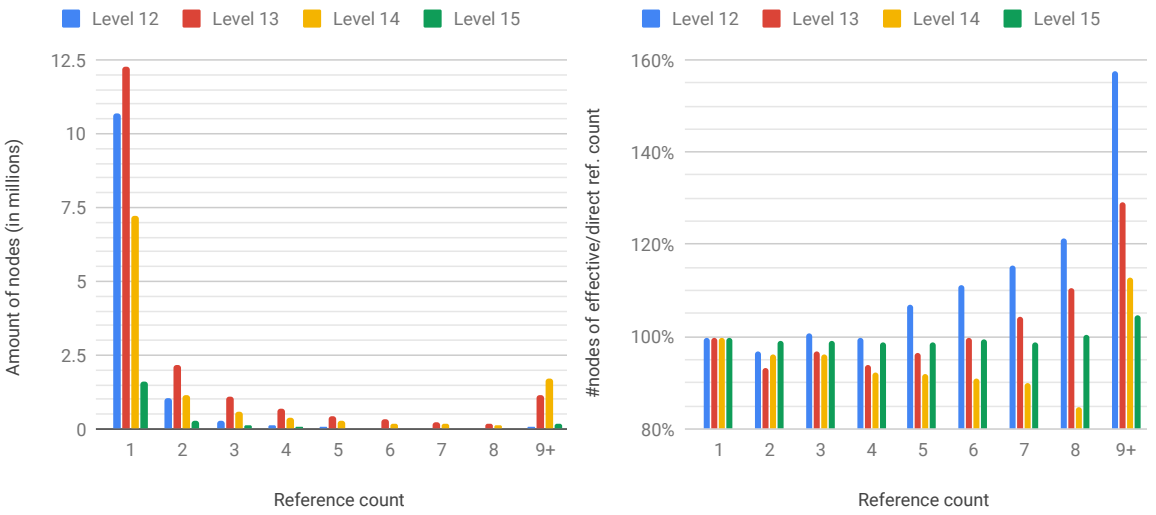


Figure 3.2: Analysis of reference counts of the Epic Citadel dataset at a resolution of $128K^3$ for the four deepest significant levels in the SVDAG. The reference counts are measured to up to eight; those with a higher reference count are added up to the 9+ category. Left: Frequency of the effective reference counts of nodes. Right: The percentage of nodes with an effective reference count relative to the direct reference count.

To give an indication of where these infrequently referenced nodes can be found in a scene, a few scenes where these nodes are highlighted are shown in Figure 3.3. This shows that these nodes are mostly present in non-uniform areas, such as edges, curved surfaces and foliage.

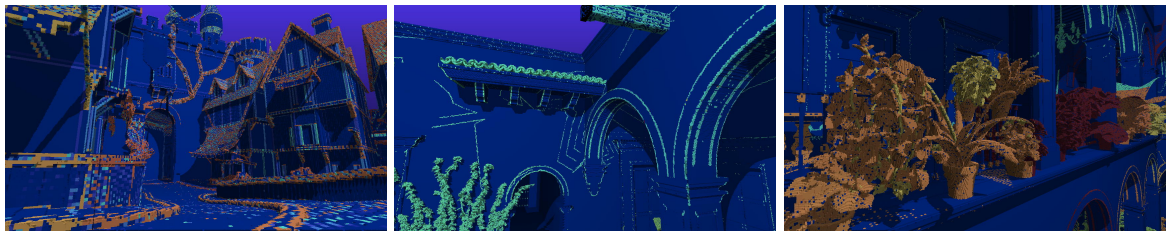


Figure 3.3: Screenshots of the Epic Citadel (left) and San Miguel (middle, right) datasets, where the nodes are colored using the Turbo colormap [Mik19] based on their reference count, where dark blue indicates a high reference count and red a low one.

Only clustering nodes that are infrequently referenced introduces a parameter to our lossy compression technique; a threshold that decides which nodes are included in the compression scheme based on their effective reference count. Nodes above this threshold are completely excluded and left intact as how they reside in SVDAG. The default value for this parameter is one, which usually includes around 60 to 70% of all nodes in the lower levels of the graph. Nodes that are referenced twice usually attribute up to 10% of all nodes in the lower levels of the graph, and nodes referenced thrice up to 5%. Increasing this parameter will increase the compression that can be achieved, at higher costs in the error rate that is produced.

3.3.2. Preventing cascading errors

Our technique is designed to run in a bottom-up approach, clustering each level individually starting at the level above the leaves. This approach has a risk of cascading errors, as lossy errors are introduced by clustering lower levels which can cause them to become similar to nodes that they were previously not similar to. When clustering is performed in higher levels, these clustered nodes in the lower levels are interpreted as the ground truth, which can in turn cause the nodes in higher levels to be clustered to nodes that they were not similar to in the original graph.

A top-down approach however would have a similar problem, since the clustering decisions made in higher levels are based on the nodes of lower levels, which will change in a later iteration. For instance, when two nodes near the top of the graph are similar, this similarity is based on the voxels in the leaf level. If for one of these nodes, one of their children further down the graph is replaced by another

node, the similarity of those parent nodes will no longer be the same as before.

We opted for a bottom-up approach, as these cascading artifacts can be easily avoided by keeping track of which nodes are parents of a cluster representative, and excluding those from the clustering process. The effect this has on the compression performance is negligible, since we only cluster nodes that have a low effective reference count.

3.4. Similar node identification

Finding identical nodes belonging to a level of the graph, which is part of the DAG algorithm, is relatively straightforward. In the bottom-up approach, the indices of the children of each node can be used to sort them. In the top-down approach, identical nodes can be efficiently found by comparing their hash values. It turns out that identifying nodes that are likely to be similar by our definition can be performed efficiently as well through a slight alteration of either of these approaches.

The problem of efficiently identifying similar nodes was also encountered by Scandolo et al. [SBE16b] for finding nodes with overlapping shadow intervals for the compression of multi-resolution shadow maps. They exploited the fact that similar nodes have an identical subtree topology. The topology of a subtree can be seen as the inner nodes of that subtree, or in other words, the same subtree excluding the nodes in the leaf level. The amount of candidate matches for each node that it needs to be compared to is limited by filtering out nodes with different topologies, which can be efficiently performed by encoding the topology of each node in a hash value. Computing hash values for efficiently identifying nodes was also employed in the top-down SVDAG construction method [KSDA16]. In both approaches, the hash values are based on the children bitmask for leaf nodes and on the hashes of all children for inner nodes. We employ a similar strategy for finding similar nodes for the goal of our lossy compression technique. For the nodes at the level above the leaves we employ a different strategy.

Inner nodes Candidate matches for the inner nodes are found by comparing their hash values, which are based on the hashes of all of their direct children. As the content of the leaf level is not a part of the topology of a subtree, the hash value for nodes above the leaf level is purely based on their own children bitmask. This approach is illustrated in Figure 3.4.

After finding a set of candidate matches for a node, the exact difference in the amount of voxels at the leaf level between the node and each of its candidate matches is found through a deep comparison of the two subtrees.

The deep comparison used to find the exact difference between two nodes is relatively costly and could be avoided by keeping track of the similarity between nodes at each level. However, it would come at the cost of a higher memory consumption. By keeping track of the similarity between nodes at earlier levels, the similarity between the children of two nodes A and B can be immediately found by checking whether a child of A is present in the set of similar nodes of the child of B. We found that this optimization is not required to find all node similarities in a reasonable time-frame, though it could be a beneficial improvement when dealing with an extreme amount of similar nodes.

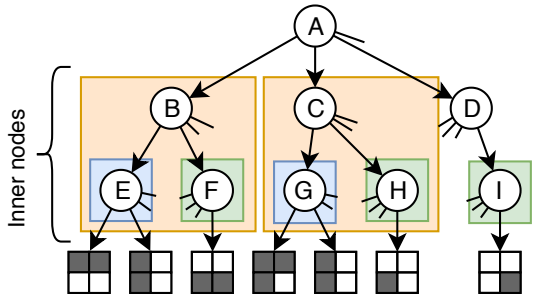


Figure 3.4: An SVO where inner nodes with identical topologies are surrounded by a box of the same color. Only nodes with identical topologies need to be compared, so this reduces the total amount of comparisons that need to be performed. Compared to what is shown in Figure 3.1, only the differences in the sets of nodes $\{E, G\}$, $\{F, H, I\}$ and $\{B, C\}$ need to be computed.

Leaf parents Filtering nodes based on their topology at the level above the leaves does not enhance the performance enough for efficiently finding similar nodes at high resolutions. Since there are only

up to eight children per node that the topology can be based on, this would result in a maximum of 255 possible unique topologies. When dealing with tens of millions of nodes at this level, which is not uncommon at high resolutions, the amount of candidate matches would still be in the range of tens of thousands. Computing the difference between each node and all of its candidate matches would not be feasible.

Fortunately, it is simple to find nodes that differ by a single voxel from each other using an alternative optimization, based on the property that nodes at this level represent at most 64 voxels. For each node, a unique 64-bit identifier can be computed as all eight 8-bit children bitmasks concatenated together into 64 bits. These identifiers are stored as the keys in a table, pointing to the index of of the node. Then, all 64 possible similar nodes for a node A that differ by one voxel can be efficiently looked up in the table. Each of these possible similar nodes should have an identifier where one of the 64 bits of the identifier of node A is flipped. This approach is illustrated in Figure 3.5. It can be straightforwardly extended for allowing nodes to differ by two or even more voxels, though it would come at a cost of more computation time, as more combinations of possible similar identifiers need to be looked up.

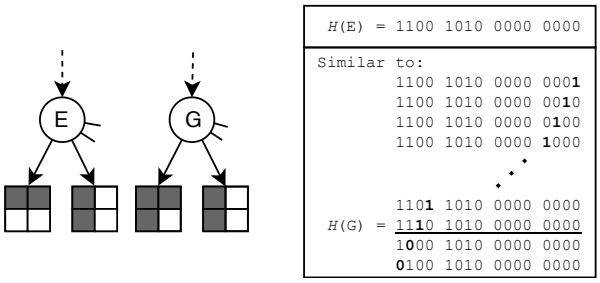


Figure 3.5: An optimization for finding similar nodes at the level above the leaves. Left: Two nodes E and G that are reside in the level above the leaves. Right: For node E, a unique identifier $H(E)$ is computed as its four children bitmasks concatenated together. Below $H(E)$, all 16 identifiers are listed that differ by one bit where the bit that is changed is marked in bold. These are used to efficiently look up nodes that differ by one voxel, such as the identifier $H(G)$ of node G. Note that in an octree, there would be identifiers consisting of 64 bits, as each node has eight 8-bit children bitmasks.

3.5. Clustering

Once all similar nodes are identified for each node, the total amount of nodes can be reduced by clustering the similar nodes together. When two nodes are similar to each other, they can be clustered by choosing one of them as a representative, and replacing all pointers that point to the omitted one to the chosen one. However, a node might be similar to many nodes, which in turn can be similar to many more nodes. Finding the best clusters in such a situation is not trivial. Luckily, this is a well-studied problem.

By modeling the similarity between nodes as a graph, a clustering algorithm can detect clusters of nodes of which it thinks are the most similar to each other. For each of the detected clusters, a single representative node can be chosen, which can be used to replace all other nodes of its cluster in the SVDAG.

3.5.1. Node similarity graph

Each pair of similar nodes can be modeled as an edge in a weighted undirected graph, where its weight represents the similarity between the two nodes. This type of graph is a representation that is used to model many types of data, such as biological gene expression networks and connections in social networks. For those types of data, detecting clusters is a commonly used approach for performing data analysis, but it is used for data compression as well.

For our purpose, the similarity s between two nodes can be defined as a function of the difference d between them and the maximum allowed difference m for them to be considered as similar: $s = 1-d/m$. The definitions of the difference d and maximum allowed difference m are provided in Section 3.2. This equation for s linearly interpolates the difference between 1 and 0. Other interpolations could be used for penalizing the clustering of nodes that have a low similarity, such as the inverse of their difference.

3.5.2. Cluster detection

For our experiments, cluster identification is performed using the Markov Clustering algorithm, using an implementation named MCL [Don00]. Our main motivation for this decision is its run-time performance, which is described in more detail in Section 4.5. Compared to the output of other clustering algorithms, we found that the detected clusters are generally very similar. A comparison of MCL and the Louvain algorithm [BGLL08] for an arbitrary graph with various granularities is shown in Figure 3.6. The granularity parameter decides how fine or coarse grained the clusters are detected at.

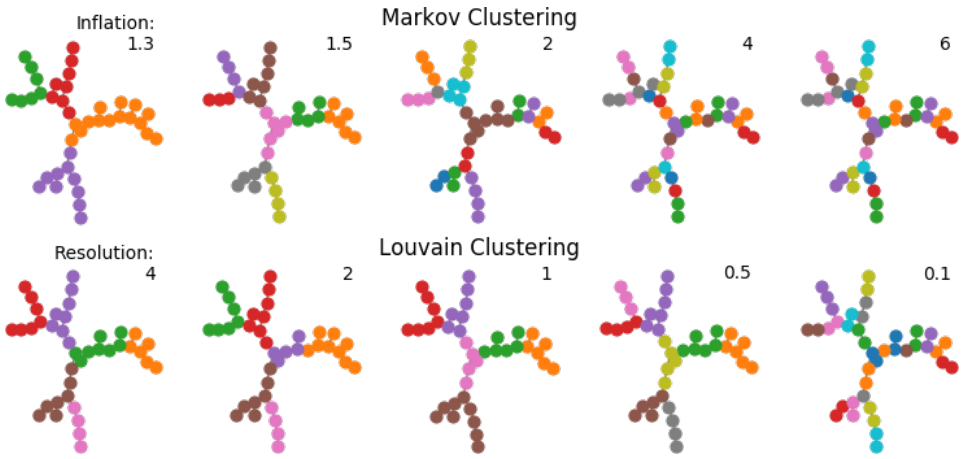


Figure 3.6: A comparison of the Markov and Louvain clustering algorithms for an arbitrary graph of node similarities encountered in one of the datasets from our experiments. In this particular case, all edge weights are 1. The parameter that decides the granularity of clusters for each algorithm is adjusted for each sub-plot, shown in the top right. The clusters are indicated with the color of the nodes.

A characteristic of Markov Clustering that we noticed is its behavior when dealing with nodes that have a single edge to another node, which we refer to as outliers. Markov clustering tends to favor clustering outlier nodes into small clusters of two or three nodes, while modularity based clustering algorithms often include the outliers in larger clusters. This is visible in Figure 3.6 for most granularities; the amount of clusters of size 3 or lower is generally quite high. This behavior of Markov Clustering is favorable for minimizing the error rate that is introduced.

3.5.3. Cluster representatives

After a set of clusters has been identified, the cluster representative for each cluster is chosen as the node in the cluster with the highest similarity to all other nodes in the cluster. In case there is no clear winner, an arbitrary node with a high similarity to most nodes is chosen. The cluster representative then replaces all other nodes in the cluster in the SVDAG by replacing all references to them in the level above to the representative. Since there is a probability that some nodes might have become identical after this process, the DAG algorithm can be performed again to merge those identical nodes together. This can reduce amount of nodes by up to 5%, depending on how many nodes have been included in the clustering process.

3.6. Compression parameters

In this chapter, we have mentioned three parameters that can be used to influence how the compression is performed.

1. **Cluster granularity** The Markov Clustering algorithm has several parameters, the most important of which is the inflation parameter which decides the granularity of the clusters. This can be used to control the amount of compression. The recommended range of this parameter is 1.2 to 6, defaulting to 2, where high values tend to produce fine grained clusters and low values produce coarse grained clusters. What this means for the compression is that a low inflation value produces a lower error and a lower compression ratio, as the amount of nodes per cluster will decrease, and vice versa. The effect of changing this parameter is highly coupled to the next parameter.

2. **Maximum allowed difference** The second parameter of our compression technique decides the maximum allowed difference between two subtrees to consider them as similar, which in essence controls the amount of edges in the node similarity graph. The parameter that controls this function is a linear scaling factor of the difference function described in Section 3.2.

3. **Reference count threshold** A third parameter is used to decide which nodes to include in the clustering stage. By default, only nodes that are referenced once are included. Choosing to include nodes that are referenced more frequently than once can moderately raise the amount of nodes that are included, at the cost of a higher loss in geometric loss and perceptual quality, since they appear more frequently in the scene.

4

Implementation

The source code of the Symmetry-aware SVDAG was used as a foundation for this thesis work, whose authors kindly made a simplified version of their implementation publicly available [VMG16]. It includes an implementation for out-of-core voxelization of triangle meshes into an SVO, an implementation of the SVDAG algorithm as described by Kämpe et al. [KSA13] and an implementation of their own SSVDAG data structure which integrates pointer compression. Additionally, it includes a viewer application that can visualize the encoded data structures in real time by efficiently traversing through the graph using the raycasting technique. It has been implemented in C++ and some crucial sections have been parallelized using OpenMP. This section provides an overview of these implementations by Villanueva et al. A more thorough description can be found in their own research work [VMG16]. Which parts have been extended for our technique are explicitly mentioned. Our source code is available as supplemental material [1].

4.1. Construction

The SVDAG is constructed out-of-core in an amount of user-specified build steps, that can be estimated based on the output resolution and the complexity of the model. In each step, a portion of the SVO is built through the voxelization algorithm and reduced to an SVDAG. The amount of steps can be increased to lower the memory consumption of the construction process, at a trade-off in computation time. These portions are progressively merged together into the final SVDAG. The voxelization step is performed on the CPU that builds an SVO in a streaming pass over all the triangles in a subvolume of space.

4.2. Memory representation

During construction, the SVDAG is stored in memory as a list of nodes per level. Each node contains a bitmask of 8 bits that indicates the existence for each of its children, and up to eight 32-bit child-pointers, which point to the to a node in the list of nodes for the next level. At the deepest level of the graph, the nodes contain no child pointers, as the bitmask by itself indicates which children exist.

The encoded memory representation of the SVDAG that is stored on disk and uploaded to the GPU is different from that of the ESVDAG, the representation used after applying pointer compression:

SVDAG When encoding the conventional SVDAG, the nodes of all levels are merged into one single list of 32 bit values. An overview of the memory layout of a single node is illustrated in Figure 4.1. The children-bitmask of each node has to be padded to 32-bits in this case, which wastes some memory. Each of the 32-bit child pointers is placed after the index where the bitmask is placed. The pointers are updated so that they point to an absolute index in this list instead of a pointer relative to the beginning of the level.

[1] https://github.com/RvanderLaan/SymVox

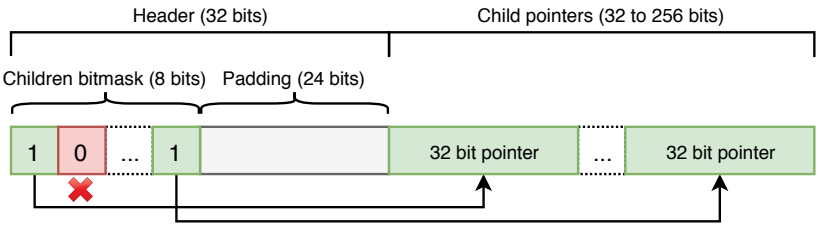


Figure 4.1: The memory layout of an inner node in the SVDAG.

SVDAG with pointer compression When applying pointer compression, a list of 16-bit half-words is used to represent the graph. Half-words of 16 bits are used instead of 32 bit words, as it simplifies the decoding of the compressed pointers. The nodes are merged into a single list here as well, but the child pointers contain values relative to the start of the next level. An overview of the memory layout of a single node is illustrated in Figure 4.2.

The nodes are sorted per level based on reference count, so that the most frequently referenced nodes are positioned at the start. Through this ordering, the most frequently used pointers can be represented with a fewer amount of bits. This is accomplished by assigning them one out of four types, which indicates the length of the pointer; 0, 16 or 32 bits. These types can be specified as 2 bits in the children-bitmask, which fills up the header to 16 bits. Therefore, no more padding in the header is needed. Another difference can be found in the leaf level, which is positioned one level higher so that the leaf nodes contain 2^4 voxels instead of 2^3. This is done to reduce the overhead of traversing through an extra level of nodes during rendering. It is performed in a post-processing step after the compression methods have been completed, and has a negligible effect on the memory consumption. Since the leaves in this format contain 2^4 voxels, it requires 64 bits per leaf node. Therefore, the leaves are stored in a separate list from the inner nodes, consisting of 64-bit integers.

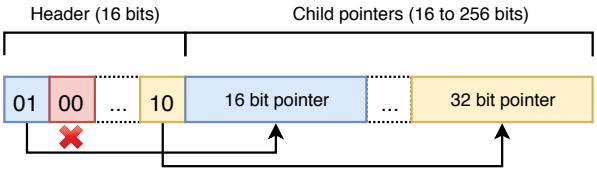


Figure 4.2: The memory layout of an inner node in the ESVDAG.

4.3. Renderer

The viewer application renders the scene with OpenGL by casting a ray for each pixel through the scene from a virtual camera. The SVDAG is traversed along points on the ray where the next node can be found until it comes across a non-empty voxel, reaches the end of the bounding box of the scene or when a maximum amount of traversal iterations is exceeded. The scene data is stored in a texture buffer object, which is essentially used as a list of integers. To find whether the voxel at a point p along the ray is empty or full, the graph is traversed from the root node to the node at position p by fetching the nodes from memory and following their pointers.

Extension We have extended the viewer application with a GUI and several debugging views for visualizing the memory location and the amount of references for each node by changing their colors. Additionally, we compute the normal direction of voxels in world-space to add diffuse lighting and hard shadows to the scene from a point light source. The normal direction of a voxel is obtained from the inverse of the direction of the last step through the voxel grid from a ray that intersects with a voxel [AW+87]. An example of the diffuse lighting and hard shadows can be seen in Figure 1.1. To compare the difference between scenes compressed with one set of parameters to another, the scenes can be loaded at run-time while keeping the same camera location and orientation.

4.4. Similar node identification

The identification of similar nodes was one of the main bottlenecks of our implementation. To optimize the computation of hash values, the hash values are computed bottom-up so that the hash values of one level can be reused in the hash computations in the level above it. After the candidate nodes with identical hash values have been found, the exact difference between the nodes is computed. As the difference between a node A and node B is identical to the difference between node B and node A, only nodes with a higher index than the one they are compared to are included in this computation. Additionally, when computing the difference between two nodes, the comparison is aborted immediately when the maximum difference has been reached. In order to improve the computation time further, it has been parallelized using the OpenMP API.

Hash value computation The computation of hash values is implemented as a recursive function, which computes the hash value of a node given the depth of its subtree. The function is recursively called for each of the children of a node, where the depth is decremented by one. The hash values are stored as 64 bit integers, which should be enough to avoid hash collisions for an extreme amount of nodes. The hash computation is implemented as follows:

- At depth 1, the child masks of all children are concatenated into 64 bits.
- For higher depths, a hash value of a node needs to be based on all child hash values. A simple and effective approach for combining multiple hash values into a hash value of the same length is to combine them with an XOR operation ($\oplus$). We make use of the the hash function included in the standard C++ library, which we refer to as H. The equation for combining two hash values v_1 and v_2 into a single hash value v with a hash function H would be $H_2(v_1, v_2) = H(v_1) \oplus H(v_2)$. This can be applied for combining an arbitrary amount of hash values by using the result of one combination into the computation for the next combined hash value.

  An observation was made that some hashes have a large amount of collisions. After some investigation, this was attributed to the fact that identical children cancel each other out due to the XOR operation being symmetrical. This was resolved by shifting the hash by one bit before applying the XOR operation. This turns the equation into $H_2(v_1, v_2) = H(v_1) \ll 1 \oplus H(v_2)$. To make the hash function truly asymmetrical, the XOR operation can be applied one more time on the result with $H(v_1)$. This greatly decreases the amount of hash collisions to below 1%.

4.5. Clustering

Since experimentation with a multitude of clustering algorithms was necessary, our application is not tightly coupled to a particular clustering implementation. The edges that are clustered are written to a text file, where each line contains the index of the source node, that of the target node and the similarity between them as a number between 0 and 1. Having access to these files proved to be useful for analysis and the tweaking of the behavior of the cluster algorithms. The detected clusters are written back to a text file by the clustering process, where each line contains the indices of the nodes of a cluster, which is subsequently processed by our application.

Since usually not all nodes are connected to each other in the similarity graph, it can be split up into smaller unconnected subgraphs, or islands. A distribution of the sizes of these subgraphs and those of the the detected clusters is shown in Figure 4.3 for an arbitrary scene. This shows that for this particular scene and resolution, a large majority of nodes is similar to only a small set of other nodes, usually fewer than five. However, there is still a notable amount of nodes with a very high amount of similar nodes. This is a trend that was visible in other scenes and resolutions as well. Additionally, it shows that the distribution of sizes of clusters is often similar to the amount of subgraphs, meaning that many subgraphs are converted to a single cluster in their entirety, though this can be influenced by the chosen parameters. The graph on the right shows that the amount of clusters that is detected compared to the amount of subgraphs given as input does not have an identical distribution for every level. At most levels, there is only an increase in the amount of small clusters, while for level 13 the clusters that are detected are generally much larger.

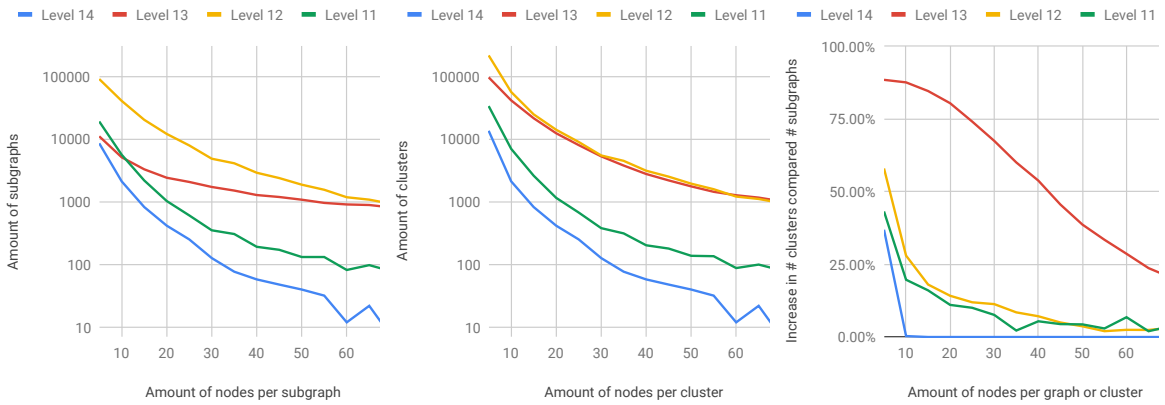


Figure 4.3: The distribution of sizes of unconnected subgraphs and clusters of similar nodes for the four deepest relevant levels of an SVDAG using the default parameters for our compression technique. Note that the first two graphs use a logarithmic scale. These were obtained from the Epic Citadel dataset at a resolution of $64K^3$. Clusters and subgraphs that contain over 70 nodes are omitted for the sake of clarity. Left: The distribution of sizes of subgraphs. Middle: The distribution of sizes of clusters. Right: The increase in amount of clusters compared to the amount of subgraphs.

Clustering the subgraphs individually instead of the whole graph at once drastically reduces the overall computation time. Due to the overhead of reading and writing the graphs to disk, small subgraphs are batched together into batches of up to $50K$ edges. Thresholds above this value did not seem to improve the overall computation time.

The clustering in our experiments is performed using MCL, an implementation of Markov Clustering by Van Dongen [Don00], which is implemented in C++ and is parallelized to make use of all available cores. While the memory consumption of MCL is low compared to other options, the computation time varied greatly. A more recent implementation of Markov clustering by Azad et al. [APO+18], focusing on high-performance parallel clustering for networks of extreme sizes, was not suitable for our use case due to its high memory consumption. We encountered similar memory-related issues for large graphs when using the Leiden and Louvain algorithm implemented in Java by Traag et al. when clustering with a low resolution parameter [TWvE19]. A drawback of MCL that we encountered is that in some instances it would get stuck after converging for up to several hours for large graphs. We were unable to pinpoint exactly why this happened, though it seemed to be related to filtering out nodes in overlapping clusters.

5 Results

Our method can be measured in terms of computation time, compression performance and the loss that is introduced. We have designed our method to apply to scenes that are voxelized using surface voxelization, meaning that the inside of solid geometry is marked as empty. A description of the datasets and their characteristics that were used in our evaluation can be found in Section 5.1.

We measure the performance of our LSVDAG using its default parameters, unless specified otherwise. These are an inflation of 2, a maximum difference factor of 1 and a maximum reference count of 1. These were chosen as they generally provide the best trade-off between compression and quality. The effect of of changing these parameters is described in Section 5.4.

Our experiments were performed on a Linux workstation with 16 GB of RAM and an AMD Ryzen 1600 processor with 6 cores (12 threads). Our LSVDAG technique can run in-core with the complete reduced SVDAG as input. The Symmetry-aware SVDAG (SSVDAG) technique can be applied on top of an LSVDAG, however the symmetry-awareness was not optimized to take advantage of the lossy similarity detection in these results. This should be feasible though, and is definitely an interesting direction for future work.

We measured no noticeable impact on rendering performance of our LSVDAG compared to that of the SVDAG with pointer compression. Compared to the original SVDAG without pointer compression, both do show a decrease in rendering performance of around 15%, as they both require a rearrangement of the nodes, which are naturally arranged in a more spatially-coherent and, hence, cache-friendly way in the original SVDAG.

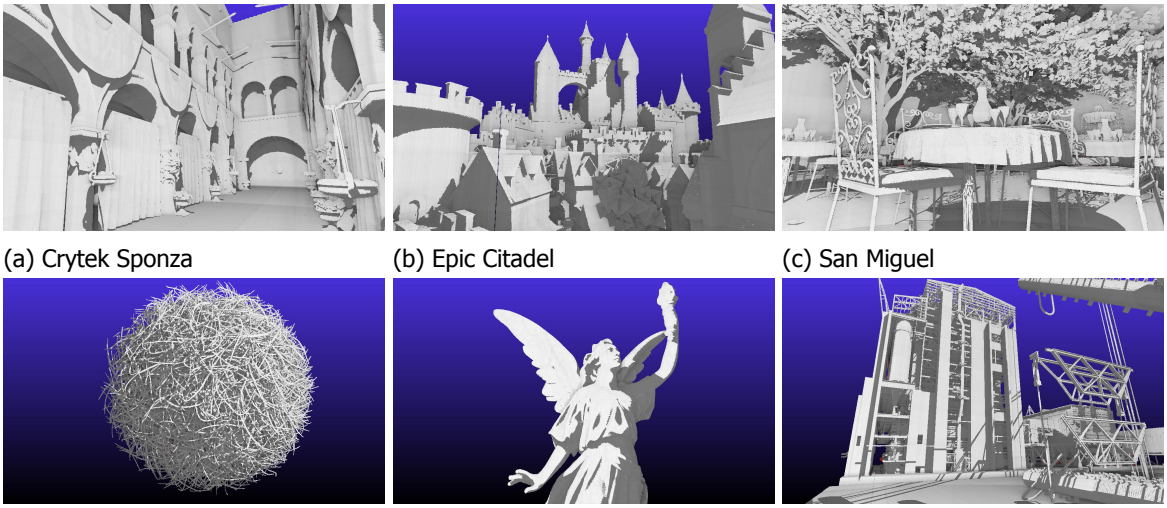
5.1. Datasets

Our method has been tested on a variety of datasets that can be directly compared to the results of the SVDAG [KSA13] and the SSVDAG [VMG16]. The first three are complex scenes representative of what can be found in modern video games. Crytek Sponza is a geometrically quite simple, Epic Citadel has local areas of high detail and San Miguel contains a large amount of foliage and a variety of fine detail. Furthermore, we tested our method on the highly irregular Hairball scene, the detailed 3D scanned Lucy sculpture and the topologically complex Powerplant CAD model[1]. Images of these datasets are shown in Figure 5.1.

5.2. Construction time

Our compression technique is designed to be performed offline, which lowers the importance of the construction time. However, it still is a useful to measure time in order to understand how well the technique scales up to higher resolutions and datasets with different kinds of complexities. Some parts of our technique have been optimized to take advantage of multi-threading, but there remain many possibilities for further optimization which were deemed unnecessary for the purpose of this research. Construction times for all datasets at varying resolutions are shown in Table 5.1 and Figure 5.2.

[1] Crytek Sponza, San Miguel and Hairball have been obtained from Morgan McGuire's Computer Graphics archive. The other datasets are courtesy of Epic Games (Epic Citadel), the Stanford 3D Scanning Repository (Lucy) and the University of North Carolina at Chapel Hill (Powerplant).



(a) Crytek Sponza (b) Epic Citadel (c) San Miguel

(d) Hairball (e) Lucy (f) Powerplant

Figure 5.1: The datasets used in our experiments, rendered in real time in our renderer application.

For low resolutions, the additional construction times are expectedly very low as well. For higher resolutions, the identification of similar nodes and especially the clustering steps take up a significant amount of time, in the order of hours in some cases. The computation time for these steps is highly coupled to the parameters that decide how many nodes are included in the clustering process.

The computation time of our technique can be broken down into three stages: Hash computation, finding similar nodes and clustering. The clustering contributes the most to the total computation

Dataset		$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$
CrySponza	MVoxels	39.6	159.0	637.3	2,552.0	10,213.1	40,873.9
	SVDAG Time	0.00	0.01	0.02	0.06	0.18	0.51
	Hash computation	0.00	0.00	0.02	0.06	0.20	0.66
	Finding similar nodes	0.00	0.02	0.13	0.61	1.98	6.22
	Clustering	0.03	0.12	0.37	1.43	5.16	16.67
Epic Citadel	MVoxels	4.5	18.0	72.2	290.3	1,166.4	4,711.6
	SVDAG Time	0.00	0.00	0.01	0.05	0.15	0.43
	Hash computation	0.00	0.00	0.01	0.05	0.16	0.47
	Finding similar nodes	0.01	0.04	0.11	0.37	1.32	4.20
	Clustering	0.03	0.09	0.36	1.06	3.34	11.29
San Miguel	MVoxels	11.8	47.5	191.9	771.0	3,093.0	12,388.8
	SVDAG Time	0.00	0.01	0.05	0.15	0.44	1.35
	Hash computation	0.00	0.01	0.03	0.13	0.52	1.69
	Finding similar nodes	0.00	0.03	0.10	0.66	4.09	15.93
	Clustering	0.06	0.18	0.97	4.87	26.22	114.49
Hairball	MVoxels	176.7	731.9	2,982.9			
	SVDAG Time	0.13	0.43	1.34			
	Hash computation	0.11	0.46	1.86			
	Finding similar nodes	0.37	2.04	14.71			
	Clustering	3.80	15.12	75.81			
Lucy	MVoxels	6.2	24.7	98.7	395.0	1,580.2	
	SVDAG Time	0.00	0.01	0.02	0.07	0.27	
	Hash computation	0.00	0.01	0.03	0.13	0.50	
	Finding similar nodes	0.00	0.03	0.32	2.70	14.90	
	Clustering	0.16	0.45	1.76	8.64	59.94	
Powerplant	MVoxels	3.9	17.0	72.5	310.5	1,336.0	5,827.0
	SVDAG Time	0.00	0.00	0.01	0.02	0.05	0.14
	Hash computation	0.00	0.00	0.00	0.01	0.04	0.14
	Finding similar nodes	0.00	0.00	0.01	0.05	0.14	0.41
	Clustering	0.02	0.04	0.10	0.22	0.68	1.92

Table 5.1: Computation time in minutes of the steps in our technique compared to the computation time of the SVDAG. The amount of non-empty voxels in millions (MVoxels) is noted for each scene to give an indication of its complexity. Some entries are missing due to hardware limitations.

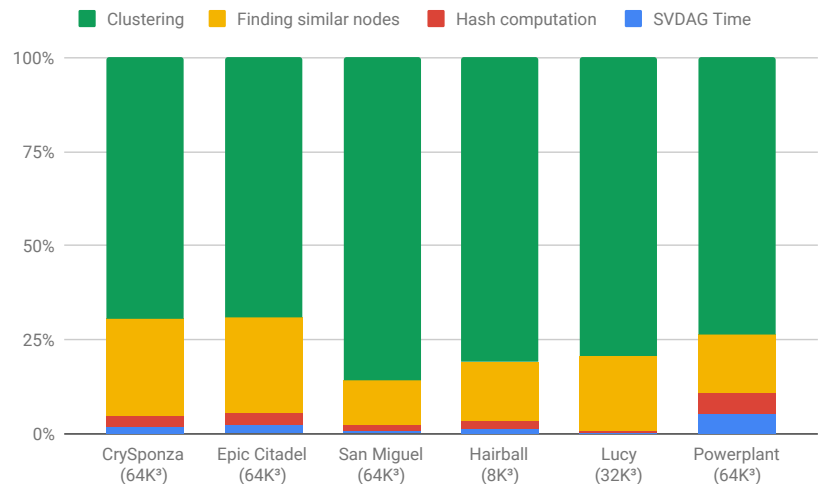


Figure 5.2: Distribution of computation time for each step at each dataset at the highest resolution they were compressed at. The time needed to perform the DAG algorithm is shown as for reference.

time, generally around 75%. The identification of similar nodes can take up to several minutes for high resolutions. The time needed to compute the hash values is the shortest, generally only a small fraction of the total computation time.

5.3. Compression performance

We measure the amount of compression as the memory usage of our LSVDAG divided by the that of the SVDAG. The same comparison is performed to the other state of the art compression methods, namely by applying pointer compression (ESVDAG) and after both applying pointer and symmetry-aware compression (SSVDAG). For reference, we also note the memory consumption of the pointerless SVO [SK06], which is equal to one byte per node.

The metric we use to measure the loss that is introduced, is the amount of voxels that differs from the original scene. This metric is determined by summing up the difference between all nodes in a cluster compared to their cluster representative, multiplied by the effective reference count of those nodes. The error rate is therefore directly related to how many clustering opportunities there are, and how similar the nodes of the clusters are.

Our raw measurements are shown in Table 5.2. A comparison of the compression ratios corresponding error rates that were achieved for all datasets is illustrated Figure 5.3. What can be seen from these results is that the effectiveness of the compression is linked to the type of the dataset that is is applied on. The compression ratio varies between 10% and 50% depending on the dataset, and the error scales up with the compression ratio from close to 0% to 3.5%.

In Figure 5.4 we compare the compression ratios of the SVDAG, ESVDAG and SSVDAG to their lossy variants. These graphs show that the effectiveness of applying additional compression methods on top of the LSVDAG is consistently less effective than applying them on the conventional SVDAG, likely due to some sort of interference.

5.4. Parameter influence

Each of the three parameters defined in Section 3.6 can be tweaked to influence the amount of compression that is applied. In this section we show the effect of these parameters on the compression ratio, error rate and perceptual quality. These metrics are shown in Figure 5.5, where for each plot describing one parameter, the others are set to default values. This shows that the inflation parameter has a marginal influence on the overall performance, while the other two have a large effect on both the compression ratio and the error rate. The Epic Citadel dataset was chosen for measuring the effect of these parameters, as it appears to be the dataset where an average amount of compression is achieved, as shown in Figure 5.3. Since the geometric error metric is not a direct indicator of the perceptual loss in quality, we provide a set of images in Figure 5.6 of the original datasets, after applying our compression method with the default parameters and with a more aggressive set of parameters.

Scene	Total number of nodes in millions						Memory consumption in MB						bits/vox
	$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$	$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$	
Crytek Sponza (0.26 MTri)													
LSVDAG	0.1	0.4	1.0	2.7	6.9	18.0	3.8	10.1	25.6	64.4	162.6	419.8	0.086
LESVDAG	″	″	″	″	″	″	1.9	5.5	15.4	42.3	115.5	317.1	0.065
LSSVDAG	0.1	0.3	0.9	2.2	5.7	14.8	1.6	4.6	12.9	35.2	95.3	257.5	0.053
SVDAG	0.2	0.5	1.3	3.5	9.1	23.6	4.3	11.9	31.7	82.3	212.7	545.3	0.112
ESVDAG	″	″	″	″	″	″	2.1	6.4	18.7	52.7	147.4	404.0	0.083
SSVDAG	0.1	0.4	1.0	2.8	7.1	18.2	1.7	5.2	15.0	41.4	112.8	304.5	0.062
SVO	12.7	52.3	211.3	848.6	3,400.6	13,613.6	12.7	52.3	211.3	848.6	3,400.6	13,613.6	2.794
Epic Citadel (0.39 MTri)													
LSVDAG	0.1	0.3	0.8	2.2	5.9	15.6	2.5	7.1	19.4	52.3	137.5	357.5	0.607
LESVDAG	″	″	″	″	″	″	1.3	3.9	11.8	34.7	97.9	269.5	0.480
LSSVDAG	0.1	0.3	0.7	1.9	5.1	13.3	1.1	3.3	9.9	29.3	82.5	228.1	0.387
SVDAG	0.1	0.4	1.0	2.9	7.6	19.9	3.0	8.5	23.8	65.4	174.7	454.7	0.772
ESVDAG	″	″	″	″	″	″	1.5	4.6	14.2	42.5	122.0	336.7	0.599
SSVDAG	0.1	0.3	0.8	2.3	6.2	16.2	1.2	3.7	11.5	34.0	97.3	268.7	0.456
SVO	1.5	5.9	23.9	96.1	386.4	1,552.8	1.5	5.9	23.9	96.1	386.4	1,552.8	2.637
San Miguel (10.0 MTri)													
LSVDAG	0.2	0.6	2.0	5.4	14.2	37.9	4.5	16.7	51.5	139.7	363.6	957.4	0.648
LESVDAG	″	″	″	″	″	″	1.9	7.5	24.8	74.4	212.5	614.8	0.416
LSSVDAG	0.2	0.6	1.8	4.8	12.5	32.6	1.8	6.9	22.3	65.6	184.9	522.3	0.354
SVDAG	0.2	0.7	2.3	6.6	18.6	50.0	4.7	18.1	57.8	164.5	460.5	1,228.2	0.832
ESVDAG	″	″	″	″	″	″	2.0	8.1	27.4	85.5	262.0	766.1	0.519
SSVDAG	0.2	0.6	1.9	5.5	15.2	40.0	1.9	7.3	23.8	72.4	212.3	603.5	0.409
SVO	3.8	15.5	63.0	254.9	1,025.9	4,118.9	3.8	15.5	63.0	254.9	1,025.9	3,928.5	2.660
Hairball (2.9 MTri)													
LSVDAG	4.7	13.8	36.8				129.9	373.4	975.3				2.743
LESVDAG	″	″	″				63.5	213.9	627.1				1.764
LSSVDAG	3.9	11.9	32.2				55.1	183.7	535.1				1.505
SVDAG	5.8	16.6	48.6				156.3	435.4	1,246.2				3.505
ESVDAG	″	″	″				72.7	240.0	782.2				2.200
SSVDAG	4.5	13.5	39.8				59.5	198.8	622.7				1.751
SVO	53.8	230.4	962.4				53.8	230.4	962.4				2.706
Lucy (28.1 MTri)													
LSVDAG	0.1	0.4	1.1	3.1	9.9		3.6	9.8	27.2	72.4	229.5		1.218
LESVDAG	″	″	″	″	″		1.9	5.9	18.3	52.8	174.6		0.927
LSSVDAG	0.1	0.3	1.0	2.8	8.6		1.4	4.7	15.1	45.1	148.3		0.787
SVDAG	0.2	0.5	1.7	5.7	18.3		4.1	11.9	38.6	133.5	439.3		2.332
ESVDAG	″	″	″	″	″		2.2	7.1	25.1	90.7	309.4		1.643
SSVDAG	0.1	0.4	1.4	4.8	14.4		1.6	5.5	20.0	70.4	223.9		1.189
SVO	2.0	8.2	32.9	131.6	526.6		2.0	8.2	32.9	131.6	526.6		2.796
Powerplant (12.7 MTri)													
LSVDAG	0.1	0.2	0.5	1.1	2.6	5.9	1.9	4.9	11.5	27.5	65.4	151.8	0.218
LESVDAG	″	″	″	″	″	″	0.8	2.4	6.2	15.7	39.6	99.2	0.143
LSSVDAG	0.1	0.2	0.4	0.9	2.1	4.8	0.7	2.0	5.2	13.2	33.0	82.0	0.118
SVDAG	0.1	0.2	0.5	1.2	2.9	7.0	2.0	5.1	12.4	30.1	73.3	175.4	0.252
ESVDAG	″	″	″	″	″	″	0.9	2.5	6.6	17.0	44.0	113.8	0.164
SSVDAG	0.1	0.2	0.4	1.0	2.3	5.4	0.7	2.1	5.5	14.0	35.4	89.9	0.129
SVO	1.1	5.0	22.0	94.4	404.9	1,741.0	1.1	5.0	22.0	94.4	404.9	1,741.0	2.506

Table 5.2: Compression performance of the proposed lossy compression technique compared to the state of the art. A set of conservative default lossy compression parameters was used to obtain these results, which are described in Section 3.6. Our LSVDAG (Lossy SVDAG), LESVDAG (Lossy ESVDAG) and LSSVDAG (Lossy SSVDAG) are compared to the conventional SVDAG, ESVDAG (SVDAG with pointer compression) and the SSVDAG (Symmetry-aware SVDAG) which were obtained through the implementation of Villanueva et al. [VMG16]. In the last column, the bits per voxel metric is specified for the highest resolution at which the dataset was processed. Some entries are missing due to hardware limitations.

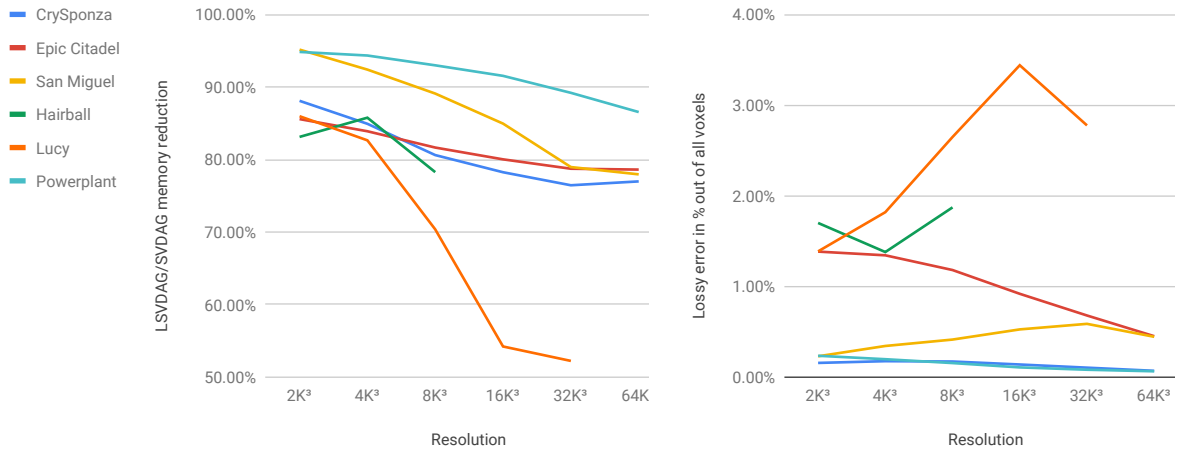


Figure 5.3: Left: Memory reduction of the LSVDAG compared to the SVDAG. Right: Error rate of the LSVDAG.

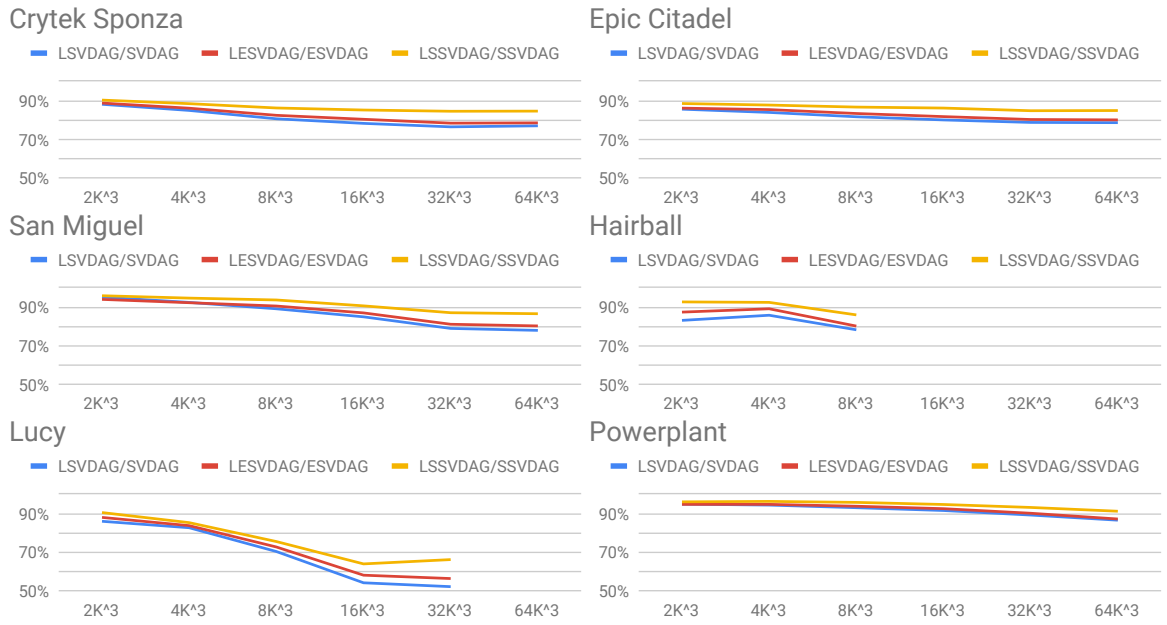


Figure 5.4: A comparison of memory consumption of the SVDAG, ESVDAG and SSVDAG to their lossy variants.

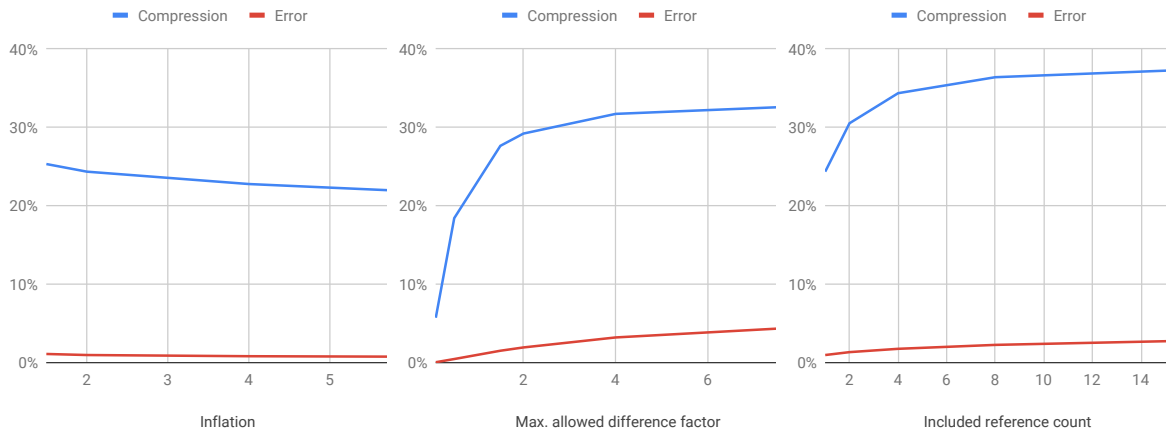
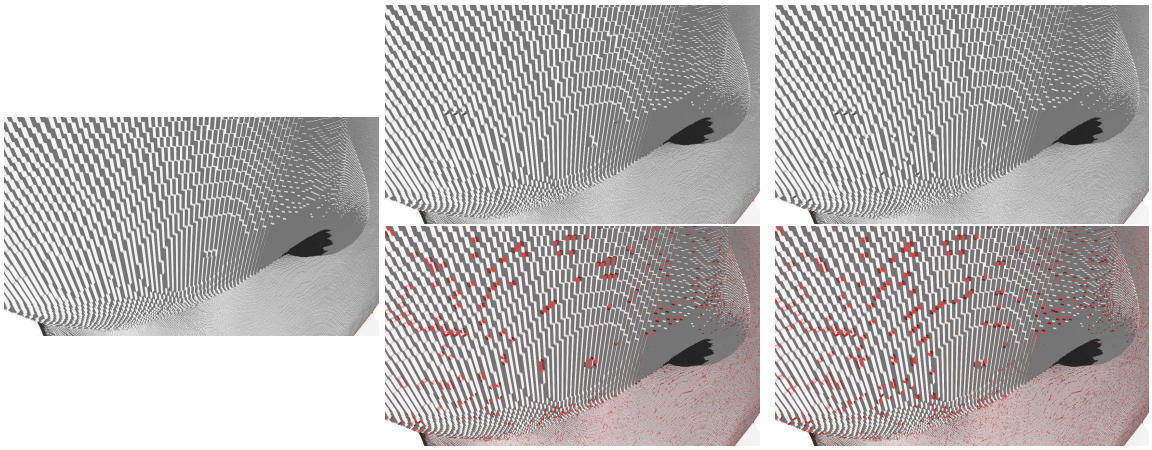
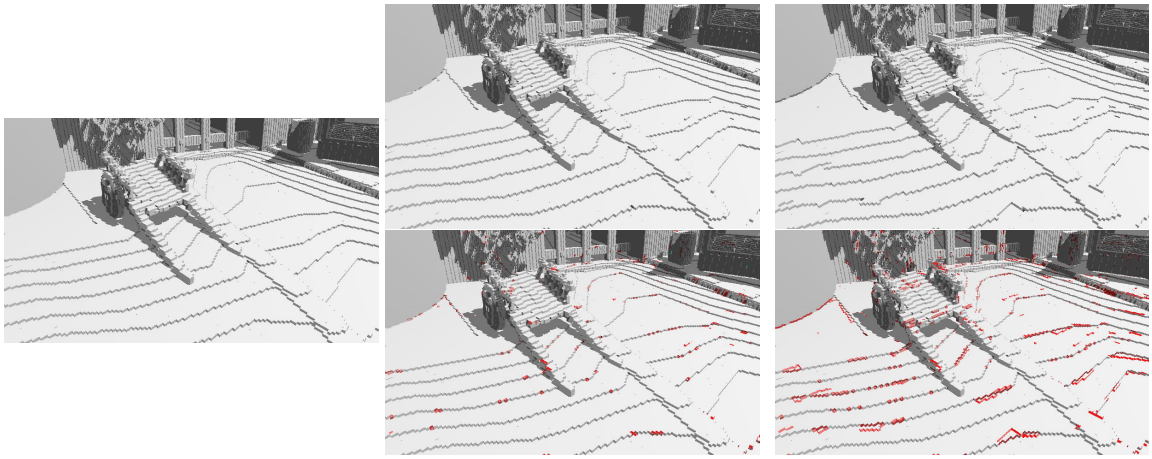


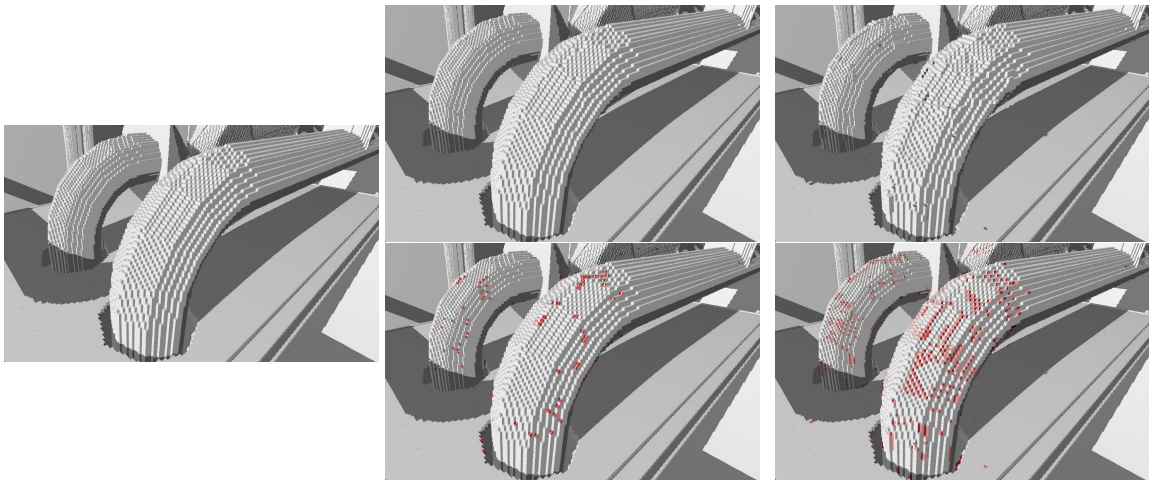
Figure 5.5: The influence of changing the lossy compression parameters. In each figure, one of the parameters is changed while the others are kept at default. The percentage on the Y-axis indicates the amount of compression in terms of memory consumption. The data of these graphs was obtained from the Epic Citadel scene at a resolution of $32K^3$.



(a) Original (439.3 MB) (b) Default (229.5 MB, 3.59% error) (c) High (178.9 MB, 4.87% error)



(d) Original (174.7 MB) (e) Default (137.5 MB, 0.95% error) (f) High (102.2 MB, 5.4% error)



(g) Original (175.4 MB) (h) Default (151.8 MB, 0.09% error) (i) High (117.6 MB, 1.07% error)

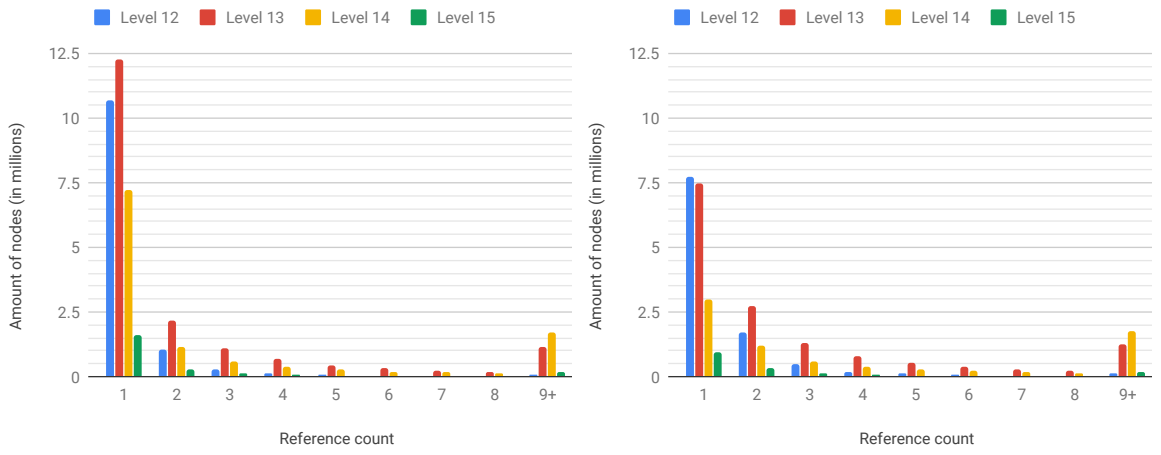
Figure 5.6: A visualization of the types of artifacts that appear. For each dataset, an image is captured of its original state (left), after applying lossy compression with default parameters (middle) and with high compression parameters (right), where the inflation is set to 1.5, the maximum difference factor to 4 and the maximum included reference count to 4 as well. The images in the middle and right at the bottom show the voxels that have changed in red. These images were captured from the Lucy dataset at a resolution of $32K^3$, the Epic Citadel scene at a resolution of $32K^3$ and the Powerplant dataset at $64K^3$. The memory consumption is measured from the LSVDAG format; without additional types of compression.

6

Discussion

From our results, we observe the following characteristics of the LSVDAG. With default parameters, the lossy compression generally reduces the memory consumption of the SVDAG by over 20%. Its effectiveness increases as the resolution grows for most datasets.

Applying pointer compression on the LSVDAG performs slightly worse than performing it on the SVDAG, generally between 1% and 2%. The reduction in effectiveness of pointer compression on top of our LSVDAG is in contrast with our expectation that increasing the amount of nodes that is referenced more frequently will benefit pointer compression. That the pointer compression does not increase in effectiveness can be explained by investigating the shift in reference counts, which is shown before and after lossy compression in Figure 6.1. These graphs show that the amount of frequently used pointers barely increases; the largest shift is towards nodes that are referenced twice, which are not exploited by the pointer compression. The most likely explanation for the actual reduction in effectiveness of applying pointer compression to the LSVDAG is that the reduction in the amount of nodes induces a greater reduction in the amount of pointers, which in turn reduces the maximum potential that the pointer compression can achieve.



(a) Before lossy compression (b) After lossy compression

Figure 6.1: Frequency of reference counts for nodes from the Epic Citadel dataset at a resolution of $128K^3$ before and after applying lossy compression. Compression is performed only on nodes that are referenced once, which explains the decrease in nodes in that point. The increase in nodes at the higher reference counts is caused by the cluster representatives becoming identical to each other and to the other nodes in the graph, which are merged together by a second run of the SVDAG algorithm.

Applying the SSVDAG [VMG16] technique on top of the LSVDAG continues the trend of reducing in effectiveness, as the ratio of compression between the LSSVDAG and SSVDAG is generally around 5% to 10% worse than the ratio between the LSVDAG and SVDAG. This reduction in effectiveness could be attributed to the fact that the lossy compression reduces the amount of nodes that are present when

applying the SSVDAG technique, which lowers the probability that symmetrically identical nodes can be found. Still, the LSSVDAG representation consistently outperforms all other representations in terms of memory consumption. On average, a 6.4% reduction in memory consumption is achieved by the LSSVDAG compared to that of the SSVDAG.

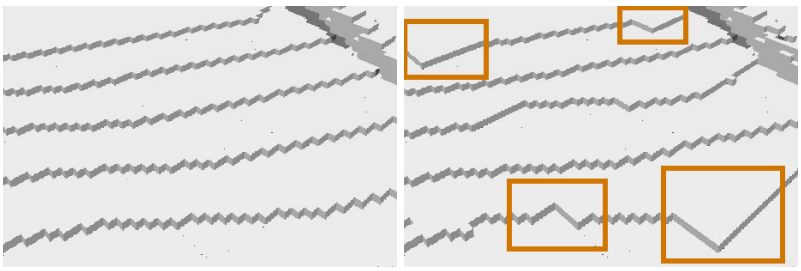
Our technique is the most effective on the Lucy dataset, with a reduction in memory cost close to 50% for a resolution of $32K^3$. This seems to be because this dataset consists mostly out of slightly curved surfaces, which causes the nodes to have similar characteristics while not being completely identical. This can be seen by the effectiveness of the conventional SVDAG; its memory consumption for the Lucy dataset is relatively close to that of the pointerless SVO. Even though the error rate is the highest for this dataset as well at 3.6%, it is not very noticeable even when viewing the model from close up, such as in Figure 5.6b.

We suspect that the compression will perform worse at higher resolutions, as this is the dataset with the highest amount of detail among our datasets. This detail can become more apparent at higher resolutions, which could cause more unique nodes to appear that likely have a fewer amount of similar nodes.

Our technique performed the worst on the Powerplant dataset. This CAD model is axis aligned and contains little high frequency detail or curved surfaces, which results in a low amount of similar nodes. This is also indicated by the high effectiveness of the conventional SVDAG on this dataset. Still, our technique is able to compress this scene by 13.5% at a resolution of $64K^3$ using default parameters at an error rate of only 0.06%.

The effect of the parameters on the compression performance is different from what we initially expected. Altering the granularity of cluster detection has a very limited effect on the overall reduction in the amount of nodes. The expected outcome when altering the granularity parameter is that it should either detect more fine-grained or more coarse-grained clusters. One reason that could explain the limited effect of this parameter is that we observed many small subgraphs to be (almost) fully connected. In those cases, often the entire subgraph is turned into a single cluster, regardless of the specified granularity. The amount of small subgraphs is generally a large majority of the total amount of subgraphs. An example of the distribution of cluster sizes is shown in Figure 4.3. One way to correct for this low influence could be to reduce the amount of edges, by either lowering the maximum allowed difference or by filtering out the edges with the lowest weights.

Some artifacts that appear as a result of our lossy compression method are more noticeable than others, which could be mitigated by changing the way the similarity between nodes is measured. In our current approach, the difference between nodes is computed as the total amount of voxels that are not identical between the leaf nodes of two subtrees. Due to the cubical shape of the volumes that that the nodes represent, an obvious artifact appears where curved edges are turned into a square ones. An example of this type of artifact is shown in Figure 6.2. To prevent this type of artifact, the similarity between subtrees could be computed in other ways. Since we only apply our compression on infrequently referenced nodes, it should be feasible to take into account the local neighborhood of nodes in the scene. For nodes of which their neighborhood is not similar, a penalty could be introduced, or the similarity between nodes could be entirely omitted to prevent those node from being clustered together.



(a) Before lossy compression (b) After lossy compression

Figure 6.2: An example of a type of square artifact that appears on slightly curved surfaces as a result of using aggressive parameters for lossy compression technique. This was captured from the Epic Citadel dataset at a resolution of $32K^3$. Left: The original scene. Right: The compressed scene, where the square artifacts are marked with an orange outline.

Another limitation that is part of the identification of similar nodes is a drawback of how it has been optimized. A topology filter is used to find candidate matches for nodes, so that only nodes that are probably similar need to be compared. However, the topology filter does not find all nodes that could be similar. When comparing two nodes, where one of them has a single voxel in the subtrees in one of its children where the other does not, the topology is not identical while they should be considered as similar. For higher levels, this is partially resolved by comparing the topologies using a shallower depth; by excluding more than one level above the leaves. As this reduces the effectiveness of the topology filter, it is a trade-off between accuracy and computation time.

The computation time of the LSVDAG can be correlated to the type of the scene in a similar way that the type of scene correlates to the amount of compression that is achieved. Since the bottleneck in computation time is the clustering step, the amount of similar nodes in a dataset that need to be processed in that step will have a large impact on the computation time. For the Powerplant scene, the overall amount of similar nodes is low, which is clearly reflected in the low computation time needed to compress it. The San Miguel scene however contains many detailed and complex objects, including a large amount of foliage. This type of geometry is often very similar, but not quite identical. Therefore, there is a higher likelihood for nodes to be similar to each other, which increases the computation time of our technique.

7

Conclusion

We have shown that the memory consumption of the SVDAG can be significantly reduced at the cost of a small loss in quality. Our lossy compression technique clusters similar nodes together through the use of a clustering algorithm that has proven its use in many application fields. The geometric error caused as part of the lossy compression is minimized by only clustering together nodes that are infrequently referenced and by only considering nodes to be similar when the difference between them is below a properly defined threshold.

The amount of compression can be influenced by changing a set of three parameters. On average, the compressed datasets take up 20.8% less memory than the original scenes using default parameters, while for more aggressive compression parameters this can increase by an additional 5% to 15%. The error rate, measured in the amount of voxels different from the original scene, is generally well below 1%. Even at higher error rates, the artifacts are only noticeable when viewing the scene from very close up. When applying additional state of the art compression methods, the relative compression improvement of our method is slightly lower due to their interference. Compared to the Symmetry-aware SVDAG [VMG16], the effectiveness of our technique is more dependent on the characteristics of the dataset it is applied on. Their approach obtains an consistent average reduction of around 20% for all datasets without applying pointer compression when compared to the SVDAG. Our approach can achieve much higher results for particular types of datasets, but generally obtains a similar compression ratio.

Our technique shares some limitations with related work. As with voxel representations in general, there is the inherit drawback of using a discretized type of geometry for surfaces that are better described using a continuous representation. Similar to the pointer compression technique, our clustering stage reorders the nodes in the graph, which affects traversal performance during rendering by around 15%. This could be reduced by finding a balance between ordering nodes based on their reference count and their spatial location.

Our method has some limitations of its own as well, that can be improved to achieve better performance. The definition of similarity between nodes does not take into account the shapes of the surfaces in the scene. This definition results in square looking artifacts, caused to the cubical shape of the volume represented by a node. It would be interesting to take the neighbourhood around a node into consideration, so that the original shape of its geometry can be better preserved. Optimizing the node similarity definition can also be further tuned to what the clustering algorithm is designed for, as for instance, a linear distribution of similarities might be interpreted differently than a logarithmic one. This optimization can also be applied to improve the loss metric that is used to measure the loss of the compression. Furthermore, the granularity parameter of the clustering algorithm has a minimal impact on the compression performance due to the characteristics of the node similarity graphs. The node similarity graphs can be altered to take more advantage of the possibilities of the clustering algorithm.

8

Future work

Besides solutions to the limitations of the LSVDAG, an interesting direction for future work would be to combine the lossy geometry compression with other compression methods, such as a lossy symmetry-awareness or a lossy cross-level merging approach, described in Appendix A. Additionally, a lossy variation of the attribute encoding as bit-trees of Appendix B might fit particularly well together with our lossy geometry compression.

In our current approach, the clustering of nodes is performed separately at every individual level, even though the nodes in the tree form a hierarchy. Another interesting direction for future work would be to consider using some sort of multi-level clustering algorithm; an algorithm that clusters the graph into several levels of clusters. This may be of use for for taking into account the nodes at multiple levels in the SVDAG at once in the clustering process.

To achieve even higher compression rates, nodes that are referenced an amount of times greater the reference count threshold could also be included in in the clustering process in some way. These nodes are likely similar to nodes that are referenced infrequently, as their high reference count implies that it is a commonly found section of geometry in the scene.

Another improvement could be made in how the cluster representatives are chosen. Currently we pick the node in the cluster that is the most similar to all other nodes in its cluster, but an entirely new node could also be generated for which the similarity to all nodes is maximized.

For the practical use of our method, it would help to automatically find an optimal set of parameters for the best quality-performance trade-off for any given particular dataset. Another useful parameter would be to set a target memory consumption value, so that a compressed dataset can be guaranteed to be below a specific size. Such a feature would be useful to ensure that a dataset will fit into GPU memory.

The identification of similar nodes has many possibilities for finding additional matches. There are completely other ways to find similar nodes compared to our current approach. For example, two nodes might look similar perceptually when all of their voxels are shifted one unit in a particular direction, while their geometric difference is relatively high.

During this research, we have come up with ideas and experimented with other ways to achieve geometry compression that have not made it into this report. One of these is a type of visibly lossless compression that can be applied on scenes with water-tight geometry, which we predict to have great potential. In these types of scenes there are areas, such as the inside of walls or statues, that can be exploited as long as they appear identical from the outside. Each child of a node can have a flag that indicates whether or not it is hidden, which can be used to ignore that child in the process for finding an identical node. Another possibility that remains unexplored is merging similar nodes together in their data representation. We believe some amount of lossless compression may be achieved by allowing each child-index of a node to optionally contain multiple pointers. For instance, when two nodes are identical except for their first child-pointer, there exists some data duplication. This could be exploited by replacing these two nodes by a single node, where the two different child-pointers can both belong to the first child index of that node. A flag in the pointer to that node can be used to indicate which of the two child-pointers to use during traversal.

A

Cross-level SVDAG

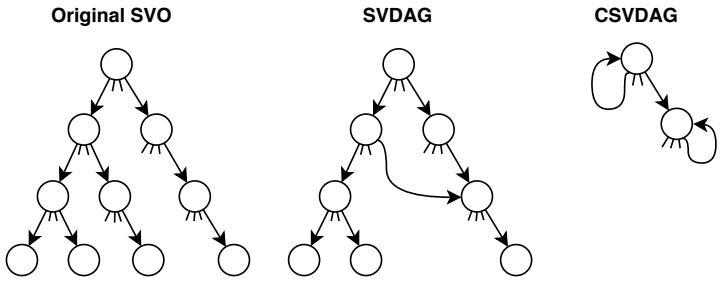


Figure A.1: An overview of our lossless Cross-level SVDAG (CSVDAG) compression technique on a quadtree. Left: The SVO used as input. Center: The SVO reduced to an SVDAG. Right: Our proposed CSVDAG technique where nodes of the SVDAG can point to nodes in the levels above them, including their own.

The SVDAG algorithm proposed by Kämpe et al only merges nodes per individual level [KSA13]. While a DAG does not allow for cycles in the graph by its definition, there should be no practical issues when a cycle is introduced, since traversal can be stopped at any desired amount of steps through the graph. The children bitmasks of a node have an identical data representation of the voxels found in leaf nodes, and can be can be interpreted as such without any modifications in the traversal algorithm used for rendering. We propose a lossless compression technique that exploits the equality of subtrees at different levels of the SVDAG. Due to the sheer amount of nodes in an SVDAG, the probability that some of these subtrees are identical is not improbable, especially for those in the deepest levels in the graph.

A.1. Method

The construction of the Cross-level SVDAG (CSVDAG) can be performed in-core after the SVDAG has been constructed. In a top-down approach, we look through the graph for identical subtrees above the level where each node resides. An example of the resulting CSVDAG after our method has been completed is illustrated in figure A.1.

We employ a hash-based approach for finding the identical subtrees at multiple levels. The hash values for all nodes at a level l are computed at a depth d, set to the full depth that subtrees have at that level. For every level above l, we compute hash values for all nodes in that level at depth d as well, so that the hash value represents the subtree of the node with the same depth as nodes in level l. Then we can directly find all identical matches and mark them as a cross-level correspondence. By merging nodes as high up in the graph as possible, it ensures that we produce a minimal DAG. A top-down approach performs faster than a bottom-up approach, since when a correspondence c is found to a node n, all of the nodes in the subtree of n also have a correspondence in the subtree of c. Once all cross-level correspondences have been identified, the duplicate nodes and their underlying subtrees are removed from the graph, and the pointers to them are replaced their respective correspondences at a higher level.

A.2. Implementation

The hash values are computed identically as was described in Section 4.4, with one exception; as the hash values are computed at different depths at every level, they cannot be reused and have to be recomputed. When nodes with identical hash values are found, a deep comparison method is used to confirm that the subtrees are identical, to prevent errors in the theoretical possibility of hash collisions.

During construction, the level of the pointers is stored in an array with an entry for each of the eight pointers, initialized as the level below where the node resides. The encoding of the SVDAG does not require any modification, as all nodes are placed in a single list where pointers can point to any node. Applying pointer compression to the SVDAG however is not compatible with this technique, as pointers to levels other than the level below the node cannot be encoded.

Pointers not only need to be updated in the level above the current level. Since nodes in lower levels may already point to nodes in the current level, those pointers need to be updated as well.

A.3. Results

The reduction in the amount of nodes depends on the type of scene and resolution. For the Epic Citadel dataset, the reduction in amount of nodes at a resolution of $64K^3$ is 18%. For the Powerplant dataset however, the reduction in amount of nodes at a resolution of $64K^3$ is 8%. The memory reduction for all datasets is illustrated in Figure A.2. The construction times are shown in Figure A.3.

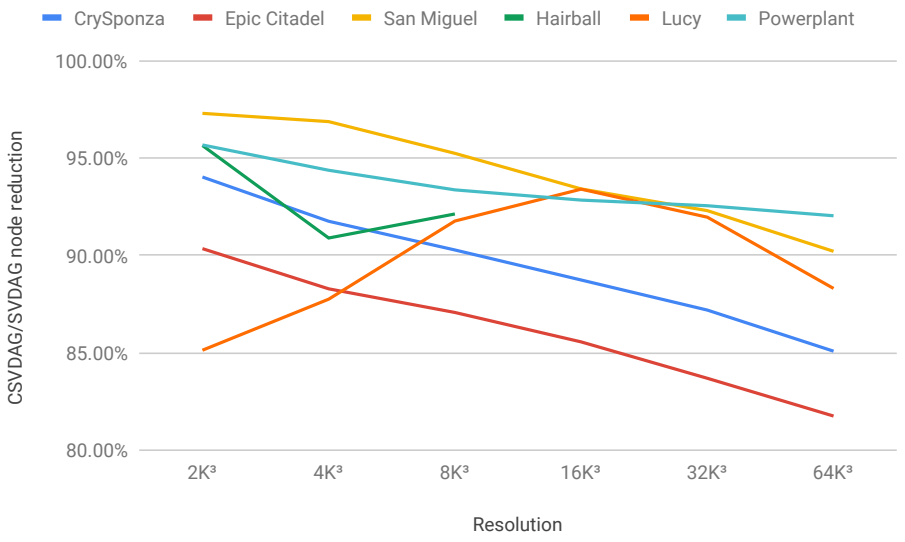


Figure A.2: Memory reduction of the CSVDAG in amount of nodes.

A side-effect of the pointers going all over the graph is a decrease in rendering performance, due to cache misses, especially on leaf level where most of the cross-level correspondences are. Memory reduction is very similar, generally up to 1% worse than node count reduction.

The fact that pointers can point to higher in the graph means that there is hidden geometry below the deepest level of the original SVDAG. This is visualized in Figure A.4. This shows that some of the nodes point back up to near to root of the graph, causing some sort of recursive detail.

A.4. Discussion

This technique is relatively simple to implement, yet quite effective. One might ponder why this has not been attempted before. This could be because it might seem unlikely that nodes on the different levels have a similar geometric structures. While this is true for nodes high up in the graph, we have shown that there is quite some overlap between nodes at levels near the bottom.

From our results, we can observe that the compression generally increases with an increase in resolution. This is likely because the amount of nodes that can be a potential match increases. For

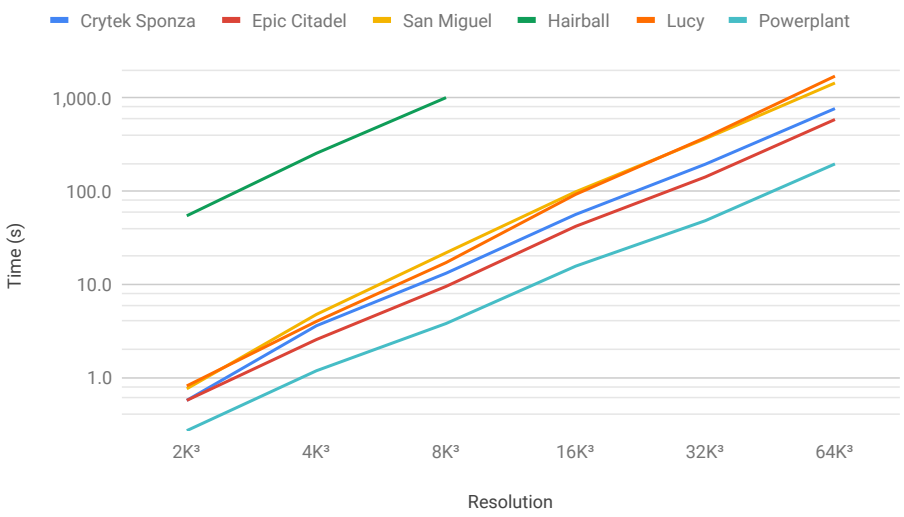


Figure A.3: Construction time of the CSVDAG in seconds, measured after the SVDAG has been constructed. Note that the time axis is logarithmic.

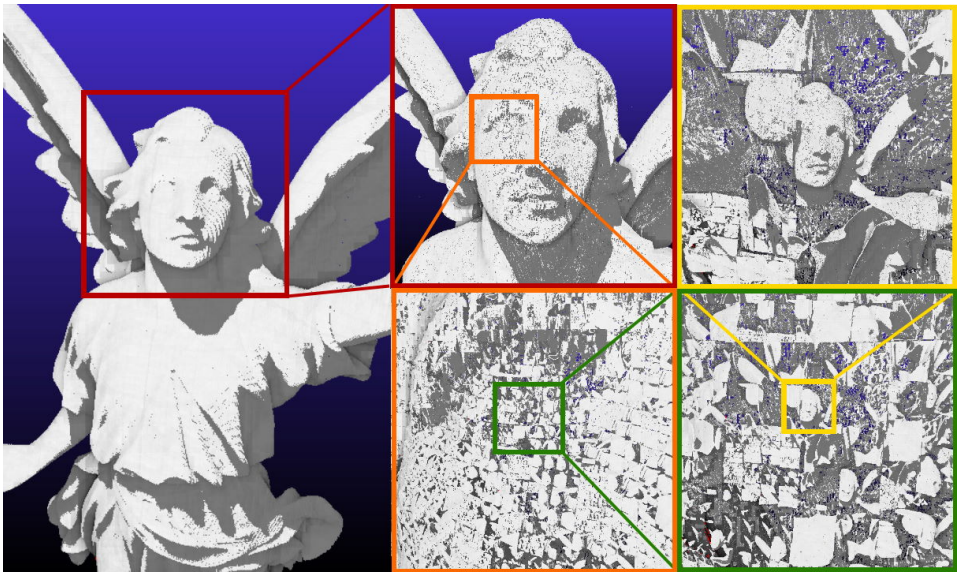


Figure A.4: A CSVDAG of the Lucy dataset, where the maximum draw level is increased above the amount of levels that are originally in the SVDAG. When looking close at the surface, earlier nodes in the graph can be seen again, such as the head.

the Lucy dataset, the compression first decreases up to a resolution of $16K^3$, after which it quickly increases with the resolution, even more than the other datasets. The cause for this initial decrease is likely related to the shape of this particular dataset, which is quite unique at every level of the graph for resolutions up to $16K^3$. Once smaller details become visible in higher resolutions, they start to become more generic and reusable at multiple levels.

The trick of traversing deeper than the maximum level of the original DAG could be consciously exploited to add a high amount of detail to geometry where this is desired, such as plants or other organic materials.

Even though the amount of nodes can be reduced, this does not directly imply a reduction in memory. The major limitation of this approach is that it is incompatible with the pointer compression technique. Perhaps a middle-ground could be reached by, for instance, limiting the pointers to one or two levels above the level they reside in, or by employing some sort of common node pool. This would affect the effectiveness of the compression and require a different construction scheme, as it would

no longer be a minimal DAG. Combining this technique with other compression methods, such as the Symmetry-aware DAG [VMG16] or our lossy compression method could significantly boost the amount of compression that is achieved.

The decrease in rendering performance could be mitigated by not applying compression to the leaf level, where the impact is not high, as there are at most 255 nodes. For the other levels, there might be a way of finding a balance between minimizing the DAG and keeping the pointers often associated with each other close together in memory.

B

Attribute bit-tree compression

In the main method of this work, we only store the presence of geometry; not their color or other material properties, such as specularity or roughness. In Efficient Sparse Voxel Octrees [LK11], even contour information was included to eliminate the blocky aesthetic, in addition to a variable amount of attachments for shading attributes. In their data structure, these shading attributes are stored uniquely per voxel in arrays separate from the geometry. Storing material properties like this in a SVDAG would greatly reduce the amount of identical subtrees, which would render it obsolete in most cases. However, material properties are often similar or equal to that of other geometry in the same region, which can be exploited. The only successful approaches up until now decouple the material properties from the geometry, so they can be compressed separately [DKB+16, DSKA17].

B.1. Method

Dado et al. [DKB+16] mention the possibility of using bittrees to store material information. Every individual bit of information can be stored in its own octree, which can afterwards be merged together into a single DAG. In other words, for n attribute bits, the attributes would be stored in a single DAG that at its root node has n children, each representing one bit of the attributes for all voxels. This is a method for storing attributes alongside the geometry, without an external data structure. As a side-experiment, a compression technique for this data structure was attempted.

Instead of encoding material information in a conventional binary encoding, it can be encoded using Gray codes. This is an ordering of numbers where two successive values only differ by a single bit. Since attributes in nearby regions of space often have similar values, such as the color of the voxels that form a leaf or a rock, we expect that the amount of bits that needs to be stored can be reduced using this encoding.

For example, in a conventional binary encoding, there is a large difference in the representation of 7 (0111) and 8 (1000); all bits are flipped, even though the values differ by one. The Gray codes for these numbers are 0100 and 1100.

B.2. Results

Due to time constraints, only small scale experiments were performed that indicated that encoding attributes as Gray codes consistently outperforms a conventional binary encoding. In Figure B.1, one of our experiments is illustrated for a simple scene containing a textured cube.

The results for more complex scenes are not mentioned here as our implementation is not sufficiently optimized for dealing with higher resolution data. The encoded attributes for complex scenes at low resolutions do not have the property that nearby voxels have similar attributes, since there the individual voxels represent a relatively large volume. When processing a smaller cut-out of a complex scene however, an improvement in memory consumption could be seen.

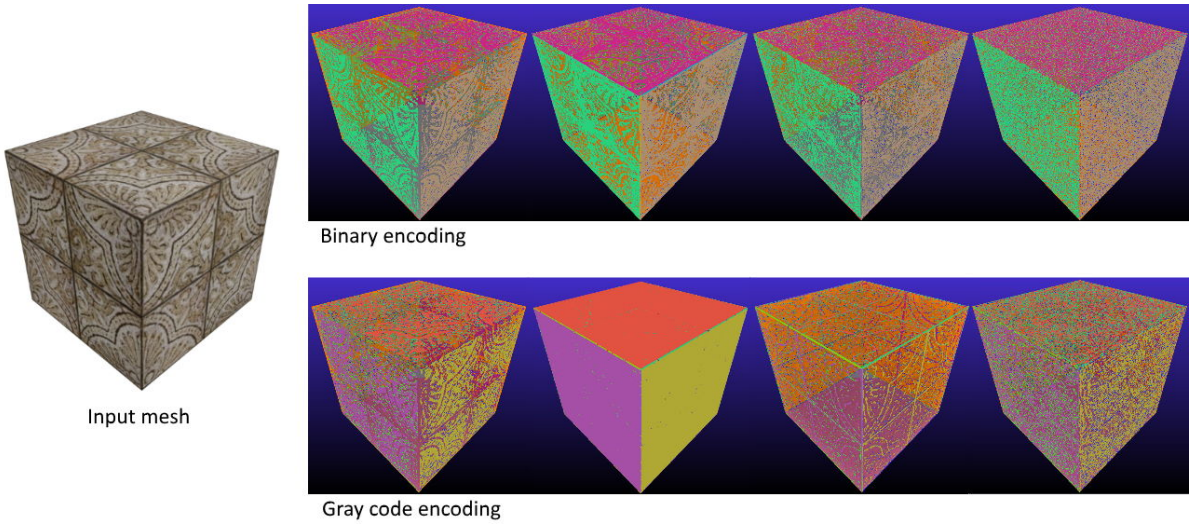


Figure B.1: A comparison of the first four attribute DAGs rendered as geometry for a simple input scene. The texture was converted to 8 bit gray-scale values, so that a total of eight attributes bits are encoded. The colors represent the memory location of each node, which makes it slightly easier to distinguish between the foreground and background, but do not add any further significant meaning.

B.3. Discussion

A small compression improvement was made by encoding attributes using Gray codes instead of the usual binary representation. For resolutions of up to $2K^3$, we observed a decrease between 1% and 7.5% in memory cost when encoding 8 attribute bits, depending on the type of scene. When comparing the individual attribute DAGs of both encodings, the binary encoded DAGs appear more noisy than the Gray code DAGs, which look significantly more regular in some instances, which causes them to be more easily exploited by the SVO and DAG algorithms.

Combining this approach for encoding attributes with our lossy compression format could potentially lead to large improvements in its memory consumption. Each of the attribute DAGs will likely include many infrequently referenced nodes that are similar to those in the other attribute DAGs, a property which our lossy compression method exploits very well.

Bibliography

[AG05] Pierre Alliez and Craig Gotsman. Recent advances in compression of 3d meshes. In *Advances in multiresolution for geometric modelling*, pages 3–26. Springer, 2005.

[APO+18] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. Hipmcl: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic acids research*, 46(6):e33–e33, 2018.

[AW+87] John Amanatides, Andrew Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10, 1987.

[BGLL08] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[BRGIG+14] M Balsa Rodríguez, Enrico Gobbetti, JA Iglesias Guitián, Maxim Makhinya, Fabio Marton, Renato Pajarola, and Susanne K Suter. State-of-the-art in compressed gpu-based direct volume rendering. In *Computer Graphics Forum*, volume 33, pages 77–100. Wiley Online Library, 2014.

[DKB+16] Bas Dado, Timothy R Kol, Pablo Bauszat, Jean-Marc Thiery, and Elmar Eisemann. Geometry and attribute compression for voxel scenes. In *Computer Graphics Forum*, volume 35, pages 397–407. Wiley Online Library, 2016.

[Don00] Stijn Dongen. A cluster algorithm for graphs. 2000.

[DSKA17] Dan Dolonius, Erik Sintorn, Viktor Kampe, and Ulf Assarsson. Compressing color data for voxelized surface geometry. *IEEE transactions on visualization and computer graphics*, 2017.

[FB07] Santo Fortunato and Marc Barthelemy. Resolution limit in community detection. *Proceedings of the national academy of sciences*, 104(1):36–41, 2007.

[For10] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[GKIS05] Stefan Gumhold, Zachi Kami, Martin Isenburg, and Hans-Peter Seidel. Predictive point-cloud compression. In *Siggraph Sketches*, page 137, 2005.

[KG00] Zachi Karni and Craig Gotsman. Spectral compression of mesh geometry. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 279–286. ACM Press/Addison-Wesley Publishing Co., 2000.

[KRB+16] Viktor Kämpe, Sverker Rasmuson, Markus Billeter, Erik Sintorn, and Ulf Assarsson. Exploiting coherence in time-varying voxel data. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 15–21. ACM, 2016.

[KSA13] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)*, 32(4):101, 2013.

[KSDA16] Viktor Kämpe, Erik Sintorn, Dan Dolonius, and Ulf Assarsson. Fast, memory-efficient construction of voxelized shadows. *IEEE transactions on visualization and computer graphics*, 22(10):2239–2248, 2016.

[LK11] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011.

[Mea82] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[Mik19] A Mikhailov. Google AI Blog Turbo, An Improved Rainbow Colormap for Visualization. https://ai.googleblog.com/2019/08/turbo-improved-rainbow-colormap-for.html, 2019. Accessed: 2019-08-20.

[MMG06] Bruce Merry, Patrick Marais, and James Gain. Compression of dense and regular point clouds. In *Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 15–20. ACM, 2006.

[New04] Mark EJ Newman. Detecting community structure in networks. *The European Physical Journal B*, 38(2):321–330, 2004.

[NVI18] NVIDIA. NVIDIA RTX™ platform. https://developer.nvidia.com/rtx, 2018. Accessed: 2019-09-27.

[RB06] Jörg Reichardt and Stefan Bornholdt. Statistical mechanics of community detection. *Physical Review E*, 74(1):016110, 2006.

[SBE16a] Leonardo Scandolo, Pablo Bauszat, and Elmar Eisemann. Compressed multiresolution hierarchies for high-quality precomputed shadows. In *Computer Graphics Forum*, volume 35, pages 331–340. Wiley Online Library, 2016.

[SBE16b] Leonardo Scandolo, Pablo Bauszat, and Elmar Eisemann. Merged multiresolution hierarchies for shadow map compression. In *Computer Graphics Forum*, volume 35, pages 383–390. Wiley Online Library, 2016.

[SK06] Ruwen Schnabel and Reinhard Klein. Octree-based point-cloud compression. *Spbg*, 6:111–120, 2006.

[SKOA14] Erik Sintorn, Viktor Kämpe, Ola Olsson, and Ulf Assarsson. Compact precomputed voxelized shadows. *ACM Transactions on Graphics (TOG)*, 33(4):150, 2014.

[SS10] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on gpus. *ACM Transactions on Graphics (TOG)*, 29(6):179, 2010.

[TOF+17] Dong Tian, Hideaki Ochimizu, Chen Feng, Robert Cohen, and Anthony Vetro. Geometric distortion metrics for point cloud compression. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 3460–3464. IEEE, 2017.

[TWvE19] Vincent A Traag, Ludo Waltman, and Nees Jan van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific reports*, 9, 2019.

[VMG16] Alberto Jaspe Villanueva, Fabio Marton, and Enrico Gobbetti. Ssvdags: symmetry-aware sparse voxel dags. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 7–14. ACM, 2016.

[Wil15] Brent Robert Williams. Moxel dags: Connecting material information to high resolution sparse voxel dags. 2015.