Lossy Geometry Compression for High Resolution Voxel Scenes

REMI VAN DER LAAN, LEONARDO SCANDOLO, and ELMAR EISEMANN, Delft University of Technology, The Netherlands

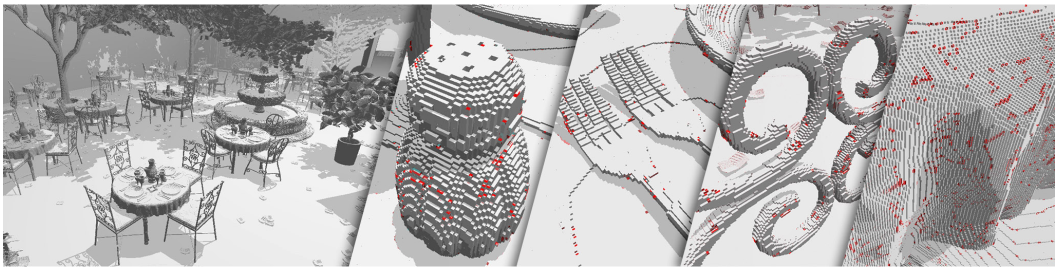


Fig. 1. The San Miguel scene at resolution $64K^3$ containing 12.4 billion voxels. Our lossy compression algorithm reduces memory requirements to store this scene as a directed acyclic graph from 1228 to 938 MB, while only changing 0.57% of all voxels. For the close-up shots, all voxels that are different from those in the original dataset are marked in red.

Sparse Voxel Directed Acyclic Graphs (SVDAGs) losslessly compress highly detailed geometry in a high-resolution binary voxel grid by identifying matching elements. This representation is suitable for high-performance real-time applications, such as free-viewpoint videos and high-resolution precomputed shadows. In this work, we introduce a lossy scheme to further decrease memory consumption by minimally modifying the underlying voxel grid to increase matches. Our method efficiently identifies groups of similar but rare subtrees in an SVDAG structure and replaces them with a single common subtree representative. We test our compression strategy on several standard voxel datasets, where we obtain memory reductions of 10% up to 50% compared to a standard SVDAG, while introducing an error (ratio of modified voxels to voxel count) of only 1% to 5%. Furthermore, we show that our method is complementary to other state of the art SVDAG optimizations, and has a negligible effect on real-time rendering performance.

CCS Concepts: • **Computing methodologies → Graphics file formats**; *Ray tracing*; • **Theory of computation → Data compression**.

Additional Key Words and Phrases: sparse voxel octree, directed acyclic graph, compression, clustering

ACM Reference Format:
Remi van der Laan, Leonardo Scandolo, and Elmar Eisemann. 2020. Lossy Geometry Compression for High Resolution Voxel Scenes. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 1 (May 2020), 13 pages. https://doi.org/10.1145/3384541

Authors' address: Remi van der Laan, r.m.vanderlaan@outlook.com; Leonardo Scandolo, l.scandolo@tudelft.nl; Elmar Eisemann, e.eisemann@tudelft.nl, Delft University of Technology, Computer Graphics and Visualization Group, Delft, The Netherlands.

1 INTRODUCTION

To render highly detailed geometry in real-time on current GPUs, novel representations alternative to the traditional rendering pipeline have received renewed interest. In particular, voxel-based approaches have a long history in computer graphics, but typically consumed massive amounts of memory for higher resolutions. Sparse voxel octrees (SVOs) hierarchically encode empty or constant space and are often found in out-of-core applications [Crassin et al. 2009, 2010; Gobbetti et al. 2008]. An extreme memory gain by orders of magnitude only became possible with the use of the Sparse Voxel Directed Acyclic Graph (DAG) [Kämpe et al. 2013], which merges identical subtrees of the SVO at every level, without affecting rendering performance. However, the effectiveness of DAG compression depends on the characteristics of the underlying data, leading to a reduced compression for very unstructured datasets.

We propose the Lossy Sparse Voxel DAG (LSVDAG), which can merge similar subtrees in addition to identical ones. Our technique exploits the fact that a large majority of the subtrees in the graph is referenced infrequently; often only once. In most scenes, these infrequently referenced subtrees are still very similar to one another. We cluster these similar subtrees and replace clusters by single representatives. The amount of compression is controllable and can lower the memory cost from 10% up to 50% compared to the original input SVDAG, while only introducing minimal errors for many different datasets.

Our main contributions are:

(1) An algorithm for efficiently finding geometrically similar subtrees in the SVDAG.
(2) A novel variable lossy geometry compression method, compatible with state-of-the-art SVDAG encodings.

2 RELATED WORK

Voxel representations have the advantage of enabling random access to any part of a scene, which can be useful for many applications. Indirect illumination [Crassin et al. 2011; Kol et al. 2019], 3D free viewpoint video [Kämpe et al. 2016a], and precomputed shadows [Kämpe et al. 2016b; Scandolo et al. 2016; Sintorn et al. 2014]. The key to making these solutions effective are an efficient encoding.

Out-of-core voxel-based approaches can handle detailed scenes via data streaming to the GPU using a sparse voxel representation [Crassin et al. 2009; Gobbetti et al. 2008]. Crassin et al. [Crassin et al. 2010] also introduces sharing of child nodes, although not for compression, but to instance objects or represent fractal structures. The introduction of Efficient Sparse Voxel Octrees (ESVOs) enabled storing a better representation of leaf-node geometry and made a step forward in terms of compression and rendering [Laine and Karras 2011]. Sparse Voxel Directed Acyclic Graphs (SVDAG) took the idea of compression further and greatly compresses high resolution SVOs losslessly [Kämpe et al. 2013] by merging identical regions. These can in fact be found abundantly in most 3D scenes and especially in binary volumes, for which it was conceived.

Some properties of the SVDAG have been exploited to further reduce its memory cost while minimally affecting traversal performance. The Symmetry-aware SVDAG (SSVDAG) [Villanueva et al. 2016] losslessly compresses the SVDAG by merging nodes that are identical through a symmetry transformation. Additionally, the authors of the SSVDAG and Dado et al. [Dado et al. 2016] independently developed another lossless compression technique that affects the size of pointers in the data structure, commonly referred to as ESVDAG. Both of the proposed techniques employ variable bit-rates for pointers based on how common a pointer is per level.

Compared to other compression techniques for 3D volumetric data, such as those described in the extensive survey by Balsa Rodríguez et al. [Balsa Rodríguez et al. 2014], a major benefit of SVDAGs

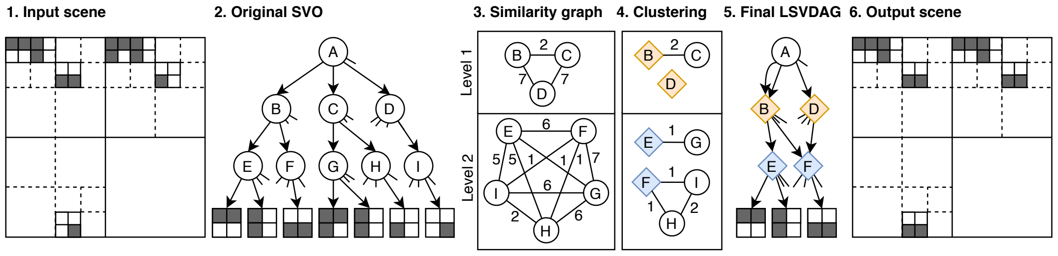


Fig. 2. An overview of our compression technique on a 2D example scene. 1) The scene used as input. 2) The Sparse Voxel Octree (SVO) constructed from the input scene. 3) A weighted undirected graph of similar subtrees is created per level, where the weights are measured as the voxel difference between two subtrees. 4) Clusters of similar nodes are identified. The cluster representatives are shown as diamond shapes; those with the least difference to all nodes in a cluster. 5) The nodes in each cluster are replaced by the cluster representatives in the SVO, resulting in our Lossy SVDAG. 6) The reconstructed output scene.

is their negligible impact on rendering performance, as there is no need for decompression. While initially limited to only representing geometry, several techniques have had success in assigning attributes to the voxels in the SVDAG [Dado et al. 2016; Dolonius et al. 2017; Williams 2015]. Lossy voxel data compression is not uncommon for volume visualization but are often aimed at compressing dense voxel grids containing attribute values, e.g., density [Bajaj et al. 2001; Ballester-Ripoll et al. 2019; Guthe et al. 2002]. Our scheme is explicitly aimed at sparse representations, which are typically also suitable for SVDAGs.

3 LOSSY SPARSE VOXEL DAGS

In this section, we outline the details of our LSVDAG construction algorithm. We will first provide an overview of our method in Sec. 3.1, followed by an explanation of our subtree similarity measure in Sec. 3.2. In Sec. 3.3 we will explain how to create a graph that globally captures subtree similarity, which will then be used to find groups of similar subtrees for merging (Sec. 3.4). Finally, in Sec. 3.5 we discuss a method for minimizing the amount of compression artifacts.

3.1 Overview

Our approach produces a lossy SVDAG (LSVDAG), where similar subtrees are replaced with a single shared representative, resulting in an SVDAG containing fewer subtrees compared to the lossless input SVDAG. Figure 2 provides a graphic overview of the steps involved in the construction of an LSVDAG structure.

A main challenge is to identify groups of similar subtrees. To this end, we model global subtree similarity per level as a graph structure, whose nodes represent subtrees of the SVDAG and edges between nodes are weighted by similarity. The similarity measure between two subtrees is based on the Hamming distance (or interchangeably, *voxel distance* or *voxel difference*) of the high resolution binary voxel grids they represent.

Once the similarity graph has been created, we look for clusters of nodes in this graph structure where intra-cluster subtree similarity is high. To this end, we employ a well-known graph clustering technique called Markov clustering [Dongen 2000].

Finally, we find a representative for each cluster with high similarity to all cluster subtrees and replace them in the original structure with this shared representation.

Clustering is performed separately for each level except for the leaf level, where lossy compression can provide little to no benefit since there are only 255 possible subtree configurations.

3.2 Subtree similarity

In an SVDAG, a subtree of height n defines a binary voxel grid of size 8^n. The distance of a pair of subtrees of equal height is given by the voxel distance of their corresponding binary voxel grids. Subtrees with a distance below a threshold are considered similar enough to be interchangeable in the final LSVDAG structure. The amount of set-voxels in surface-based geometrical data grows quadratically per subtree level [Kämpe et al. 2013], so we increase the threshold value quadratically per level, in order to consider the same amount of relative differences between subtrees of any height. In our tests, we have also experimented with a linear and a cubic function of the subtree level, but found that they respectively cluster together subtrees too infrequently or too aggressively as the height of the subtrees increases.

Formally, the *applied threshold* m is given by $m := h^2 \cdot p$, where h is the height of the subtree root and p is specified by the user, and is set to 1 by default. Lowering p will decrease the compression ratio, error rate, and computation time. We define the *subtree similarity* s between two subtrees as $s = 1 - d/(m+1)$, where d is their voxel difference.

3.3 Subtree similarity graph

To identify groups of similar subtrees, we encode the subtree similarity in a weighted undirected graph and perform a graph clustering algorithm. To explain the procedure, we start by defining the resulting graph.

Each node in this graph represents a subtree. Each edge in the graph will have a weight in form of the subtree similarity s. Subtrees with a difference above the threshold are not connected.

Directly checking all subtree pair combinations per level in an SVDAG with millions of unique subtrees would be infeasible. Instead, we only compare subtrees if we know they are likely to be similar to each other. To verify these conditions, we rely on two acceleration mechanisms; a fast topology hash and a bit encoding of height-two subtrees.

A topological hash, akin to [Kämpe et al. 2016b; Scandolo et al. 2016] allows us to efficiently find sets of subtrees with equal topology except for the bottom (leaf) level. Starting from the second lowest level and upwards, a 64-bit hash value is computed as a function of the 8-bit mask of the current subtree node and the hash values of the children, if available. Then, for each level, we compute the similarity only between subtree pairs of equal hash by counting their voxel difference in the full binary voxel grid they represent, and assume a zero similarity for all other combinations. This means that we allow differences between subtrees on the last level only, which prevents spurious unconnected voxels being introduced by our compression scheme. This approach is illustrated in Fig. 3.

In the special case that two subtrees are compared that have height two, the above hash approach would not be efficient, as we would only distinguish 64 cases, leaving us with many potential matches that would have to be verified. While possible in principle, it would take a lot of compute time. In order to maintain sensible compute times (see Sec. 4), we overrule the user specified threshold for this case and enforce a maximum of one-voxel difference. This allows us to employ a much faster scheme for trees of height two that works as follows.

A tree of height one can be encoded as an 8-bit integer, which we can concatenate to obtain a unique 64-bit encoding for trees of height two. We store all encountered encodings in a sorted list, simply ordered by interpreting the bitmask as a number. For a given subtree S, all subtrees within one voxel difference have a bitmask that is given by the bitmaks of S, in which we flip a single bit. This means that there are 64 bitmasks corresponding to subtrees that could potentially be merged. For each of the 64 cases, we search for the bitmask in the sorted list using an efficient binary search to discover potential matches. Hereby, we reduce the overall complexity from $O(N^2)$

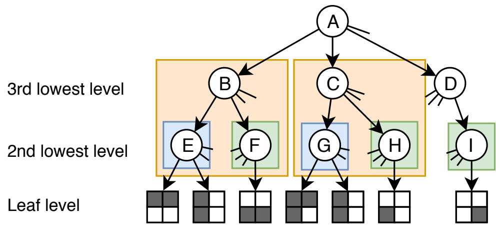
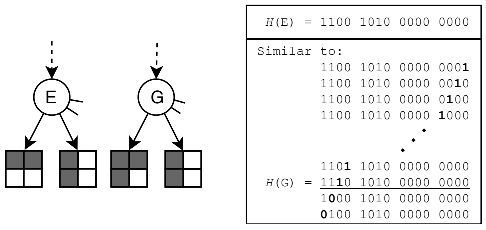


Fig. 3. An SVO where subtrees with identical topologies except for the leaf level are surrounded by a box of the same color. Only these matching subtrees need to be compared, which reduces the total amount of comparisons that need to be performed. Compared to what is shown in Figure 2, only the differences in the sets of subtrees $\{E, G\}$, $\{F, H, I\}$ for the second lowest level and $\{B, C\}$ for the third lowest level need to be computed.



```
H(E) = 1100 1010 0000 0000

Similar to:
       1100 1010 0000 0001
       1100 1010 0000 0010
       1100 1010 0000 0100
       1100 1010 0000 1000
                 ⋰
       1101 1010 0000 0000
H(G) = 1110 1010 0000 0000
       1000 1010 0000 0000
       0100 1010 0000 0000
```

Fig. 4. An example of our algorithm for finding similar subtrees at the second lowest level. Left: Two subtrees E and G of height 2. Right: For subtree E, a unique identifier $H(E)$ is computed by concatenating the bitmasks of its children. Below $H(E)$, we list all 16 identifiers created by flipping one single bit of $H(E)$. These identifiers are then used to efficiently look up subtrees that differ by one voxel from E, such as G.

to $O(Nlog(N))$, where N is the number of height-two subtrees. Fig. 4 illustrates this process. As a side product, the encoding in form of a 64 bit integer can also accelerate the comparison of taller subtrees; when reaching the second-lowest level the differences between corresponding height-two subtrees can be computed by counting the set bits after an xor-operation between their encodings.

3.4 Clustering

Having computed a similarity graph between the subtrees, we can now use this information to cluster subtrees for our lossy compression. Indeed, clusters of highly interconnected nodes in the subtree similarity graph correspond to groups of subtrees in the SVDAG that can potentially be replaced by a single representative. Nevertheless, long chains of connected nodes in the similarity graph can link two very different subtrees in the SVDAG, so finding groups of nodes that are all similar to each other is not trivial. Fortunately, this is a well-studied problem within the field of discrete mathematics, where clustering algorithms are used as a solution.

Two leading clustering algorithms are Markov Clustering [Dongen 2000] and the Louvain clustering algorithm [Blondel et al. 2008]. Fig. 5 shows a comparison of the clustering results of a small sample graph for different parameters of the algorithms, modifying the size of the detected clusters. We can observe that Markov clustering favors clustering outlier nodes (connected to a single other node) into small clusters of two or three nodes, while Louvain-based clustering algorithms often include the outliers in larger clusters. In Figure 5 for a few granularities; the amount of clusters of size 3 or lower is generally quite high. The behavior of Markov Clustering is favorable, as it minimizes the error rate that is introduced. Additionally, its run-time performance, as well as memory requirements outperform the alternative. For these reasons, we employed Markov Clustering in all examples.

Markov clustering works based on the assumption that there will be many edges between nodes within a cluster, and few edges to outside nodes. The clusters are detected by simulating and analyzing random walks through the graph, which tend to stay within a single cluster rather than move out of it, based on the prior assumption.

The computation time and memory requirements of the clustering step can be reduced by identifying smaller disjoint subgraphs, or islands, in the similarity graph, and clustering them separately. This can be done efficiently in parallel. Additionally, many of the subgraphs contain a very low amount of nodes, and can be directly turned into a cluster in trivial cases.

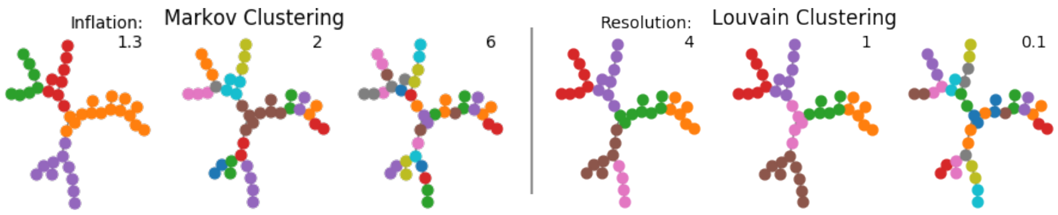


Fig. 5. A comparison of the Markov and Louvain clustering algorithms for an arbitrary graph of subtree similarities encountered in one of the datasets from our experiments. In this particular case, all edge weights are 1. The parameter that decides the granularity of clusters for each algorithm is adjusted for each sub-plot, shown in the top right. The clusters are indicated with the color of the nodes.

3.4.1 Cluster representatives. For each cluster, the node with the highest similarity sum to all other nodes (chosen at random in case of ties) is used as a representative for that cluster. At this point, we could decide to ensure that we replace only the nodes in the cluster by the representative, if their difference to this representative is below the set threshold. In practice, this has a negligible effect in the compression and error rates in all of our scenes except for the Lucy scene, which is a particular case (see Results). Here, we observe a worse compression (around 12% more data, e.g., from 216 MB to 242 MB for $32K^3$ resolution) but a slight increase in fidelity (error rate changed from 3.5% to 2.5% at $32K^3$ resolution). Given that no visible artifacts were observed, we proceeded to merge all nodes within a cluster by default.

The chosen representative then replaces all subtrees within the cluster. We update the SVDAG by changing all parent references to the chosen representation. After this step has been completed for all clusters, it is possible that subtrees at higher levels might have become identical. These subtrees are then merged using the original (lossless) SVDAG algorithm, which in our experiments can further reduce the amount of subtrees by up to 5%. This last step can either be performed after clustering each level or after clustering all levels, which leads to identical results.

3.5 Minimizing compression artifacts

To minimize compression loss and speed up compression time, we can decide to only consider infrequently referenced subtrees for the merging process, as these only occur rarely in the scene and will therefore have limited global impact. To compute how often the subtree appears in the scene, we cannot simply count the amount of times a particular subtree pointer appears in the SVDAG structure, since ancestors of that subtree may be repeated multiple times. Instead, we determine an *effective reference count* in an efficient top-down traversal of the SVDAG to derive precisely how often each subtree appears in the scene.

The distribution of effective reference counts for the bottom four levels of the Epic Citadel scene is shown in Figure 6, and all other scenes follow a similar distribution. It turns out that uniquely referenced subtrees amount to roughly 60 to 70% of all nodes in the lower levels of the SVDAG, with a sharp decrease for subtrees referenced two times and more. Therefore, choosing nodes for the similarity graph based on the effective reference count provides a good trade-off of compression and loss. In practice, even using only the nodes that occur once in the scene leads to a strong compression potential.

Fig. 7 shows two views of our test scenes that have been color coded to indicate the effective reference count, which expectedly shows the least referenced subtrees appear in curved surfaces and foliage.

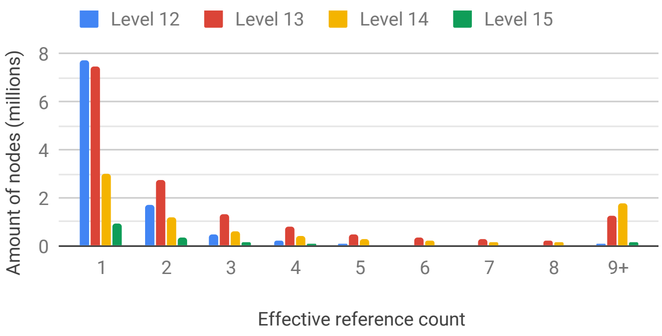


Fig. 6. Analysis of the effective reference counts of the Epic Citadel dataset at a resolution of $128K^3$ for the four deepest significant levels in the SVDAG. The reference counts are measured up to eight; those with a higher reference count are added up to the 9+ category.

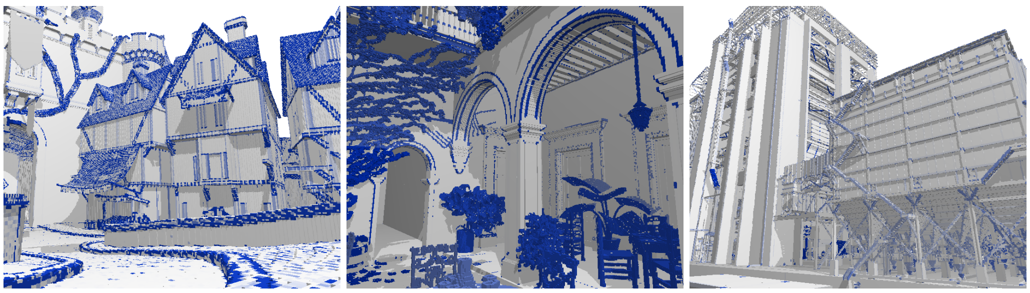


Fig. 7. Epic Citadel (left), San Miguel (middle) and Powerplant (right), where the infrequently referenced nodes are marked blue, each constructed at $64K^3$ or $128K^3$ and rendered at two or three levels lower.

4 RESULTS

The results of this section were obtained on a Linux workstation with 32 GB of RAM and an AMD Ryzen 1600 processor with 6 cores (12 threads). Our compression pipeline was implemented in C++ using MP to parallelize limited steps. As such, there is still room for optimization, possibly by performing parts of the node graph construction and clustering on a GPU. The test scenes (Fig. 8) were selected to exemplify performance for a wide range of scenes. The Crytek Sponza and San Miguel scenes are examples of 3D scenes used in interactive applications such as games or architectural visualization, with the latter displaying complex geometry in the form of tree leaves. The Lucy scene is an example of detailed geometry obtained from 3D scanning with an abundance of soft curves. In contrast, the Powerplant scene displays a large amount of sharp corners typically found in CAD plans of industrial buildings. Finally, the Hairball scene is a complex and hard to represent scene used as an extreme test case for voxelization methods. In all cases, the input SVDAG was created via surface voxelization.

Unless specifically stated, our LSVDAG construction method uses as default parameters a difference threshold factor of 1.0, and subtree maximum effective reference count of 1. The Markov clustering is controlled by one parameter, the inflation value, which we kept at 2.0. These parameters were selected empirically to provide a good trade-off between compression and quality.

Our compression technique can run in-core with the complete reduced SVDAG as input. The result of our method remains an SVDAG [Kämpe et al. 2013], and we measured no noticeable difference in terms of rendering performance.

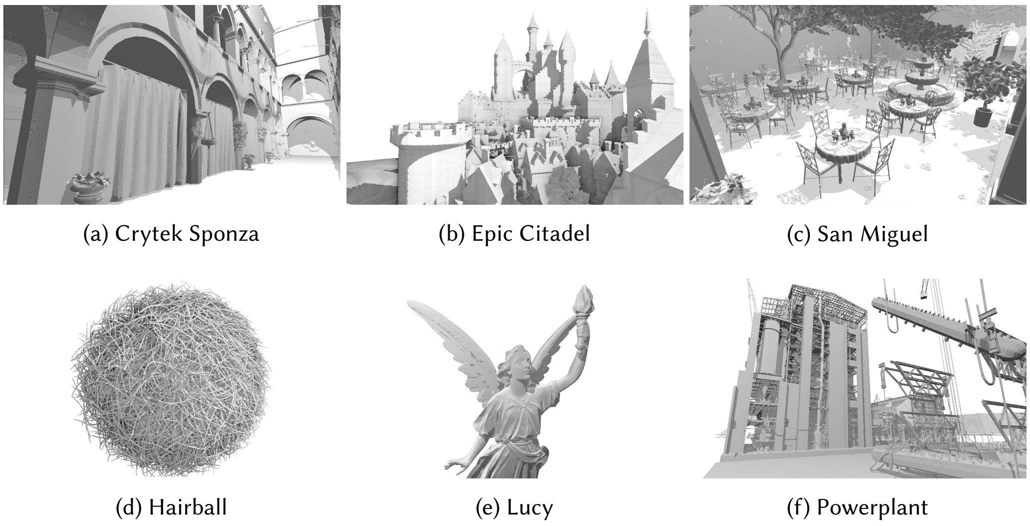


Fig. 8. The datasets used in our experiments, rendered in real time using an SVDAG renderer.

4.1 Compression times

Compression times for all scenes using standard parameters are shown in Table 1 and Fig. 9 for various resolutions. Most time is spent on clustering (roughly 75%) and finding similar nodes (roughly 20%).

For a resolution up to $4K^3$, compression times are low (in the order of seconds) for all but the Hairball scene, since in those cases the amount of subtrees is low. Fig. 9 shows that the overall computation time grows linearly relative to the resolution. There is no clear correlation between voxel count and construction time. The construction time rather depends on the scene properties, namely the overall amount of similar subtrees. As can be seen in Table 1, the total compression time for the Powerplant scene at $64K^3$ resolution involves almost 6 billion voxels but can be performed in roughly two minutes, whereas for the Lucy scene at a lower resolution, and involving 1.5 billion voxels, the compression can exceed one hour. The Powerplant scene contains mostly axis-aligned surfaces, resulting in many equal subtrees which the original SVDAG already compresses very well. Therefore, the amount of distinct subtrees included in the similarity graph is an order of magnitude lower than for the Lucy scene, explaining the lower compression times.

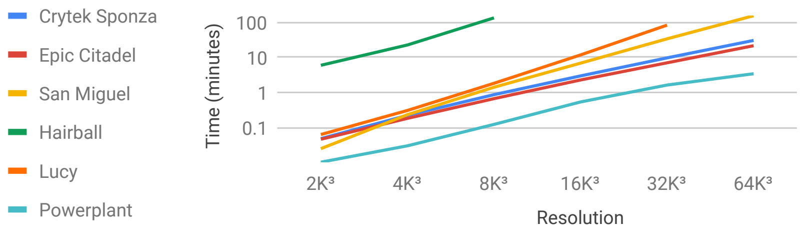


Fig. 9. Total computation time of our method for each dataset using default parameters.

		$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$
CrySponza	MVoxels	39.6	159.0	637.3	2.6K	10.2K	40.8K
	MNodes	0.04	0.13	0.41	1.16	3.18	8.17
	MEdges	0.05	0.27	1.31	5.20	16.39	44.07
	SVDAG constr.	0.00	0.01	0.02	0.06	0.18	0.51
	Sim. graph constr.	0.0	0.0	0.2	0.9	3.0	9.6
	Clustering	0.0	0.2	0.6	1.9	6.0	18.3
Epic Citadel	MVoxels	4.5	18.0	72.2	290.3	1.2K	4.7K
	MNodes	0.03	0.10	0.32	0.92	2.54	6.54
	MEdges	0.07	0.30	1.10	3.46	10.63	31.21
	SVDAG constr.	0.00	0.00	0.01	0.05	0.15	0.43
	Sim. graph constr.	0.0	0.0	0.2	0.6	2.0	6.3
	Clustering	0.0	0.1	0.5	1.6	4.6	13.8
San Miguel	MVoxels	11.8	47.5	191.9	771.0	3.1K	12.4K
	MNodes	0.02	0.11	0.45	1.70	5.86	16.35
	MEdges	0.03	0.31	2.00	12.71	62.44	207.94
	SVDAG constr.	0.00	0.01	0.05	0.15	0.44	1.35
	Sim. graph constr.	0.0	0.0	0.1	1.0	6.1	25.9
	Clustering	0.0	0.2	1.2	5.7	26.9	121.4
Hairball	MVoxels	176.7	731.9	3.0K			
	MNodes	1.51	4.04	16.07			
	MEdges	10.32	48.52	226.58			
	SVDAG constr.	0.13	0.43	1.34			
	Sim. graph constr.	0.5	3.2	24.1			
	Clustering	5.3	18.8	103.5			
Lucy	MVoxels	6.2	24.7	98.7	395.0	1.5K	
	MNodes	0.04	0.18	0.81	3.18	9.55	
	MEdges	0.11	0.81	5.84	40.09	218.13	
	SVDAG constr.	0.00	0.01	0.02	0.07	0.27	
	Sim. graph constr.	0.0	0.0	0.5	3.9	21.5	
	Clustering	0.1	0.2	1.3	7.6	56.9	
Powerplant	MVoxels	3.9	17.0	72.5	310.5	1.3K	5.8K
	MNodes	0.01	0.03	0.07	0.20	0.56	1.63
	MEdges	0.01	0.03	0.10	0.35	1.27	4.54
	SVDAG constr.	0.00	0.00	0.01	0.02	0.05	0.14
	Sim. graph constr.	0.0	0.0	0.0	0.1	0.3	0.3
	Clustering	0.0	0.0	0.1	0.4	1.2	2.8

Table 1. Voxel and similarity node/edge counts in millions and computation times in minutes of the steps in our technique and the original SVDAG creation. Some entries are missing due to hardware limitations.

4.2 Compression rates

Table 2 compares the structure size of our output to that of the original SVDAG structure. For reference, we also include the memory consumption of a pointerless SVO [Schnabel and Klein 2006], which is equal to one byte per SVO tree node. Fig. 10 provides a graphical visualization of the compression rates and corresponding error rates in our test scenes. Error is defined as the voxel difference of the original and the modified binary voxel grids divided by the count of set-voxels in the original grid.

Our technique shows the best compression rate on the Lucy scene. This is because the scene contains many low-frequency curved surfaces in every orientation, leading to many similar but non-equal subtrees, which can be captured by our LSVDAG compression but not by the original SVDAG algorithm. Conversely, the Powerplant scene exhibits the worst results. This scene contains mostly axis-aligned surfaces, which the original SVDAG already compresses very well since the common alignment results in a large amount of repeated subtrees. However, this means that little similarity can be found among subtrees corresponding to differently-aligned surfaces. This is visible

		Total number of DAG nodes in millions						Memory consumption in MB						bits/vox
Scene	Method	$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$	$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$	
Crytek Sponza (0.26 MTri)	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
	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
	LSVDAG	0.1	0.4	1.0	2.6	6.6	17.1	3.8	9.9	24.9	62.2	156.0	400.2	0.082
Epic Citadel (0.39 MTri)	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
	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
	LSVDAG	0.1	0.3	0.8	2.2	5.7	14.9	2.5	7.0	18.9	50.4	132.2	343.1	0.611
San Miguel (10.0 MTri)	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
	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
	LSVDAG	0.2	0.6	2.0	5.3	14.0	36.7	4.5	16.8	51.8	139.2	360.3	938.3	0.635
Hairball (2.9 MTri)	SVO	53.8	230.4	962.4				53.8	230.4	962.4				2.706
	SVDAG	5.8	16.6	48.6				156.3	435.4	1,246.2				3.505
	LSVDAG	4.8	13.7	35.6				132.7	373.3	951.1				2.675
Lucy (28.1 MTri)	SVO	2.0	8.2	32.9	131.6	526.6		2.0	8.2	32.9	131.6	526.6		2.796
	SVDAG	0.2	0.5	1.7	5.7	18.3		4.1	11.9	38.6	133.5	439.3		2.332
	LSVDAG	0.1	0.4	1.0	2.8	9.3		3.5	9.2	23.8	65.2	216.2		1.148
Powerplant (12.7 MTri)	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
	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
	LSVDAG	0.1	0.2	0.5	1.1	2.5	5.8	1.9	4.8	11.5	27.3	64.8	149.3	0.215

Table 2. Structure size of our structure output with defaults parameters compared to the input SVDAG. Bits per voxel metric is specified for the highest resolution at which the dataset was processed. Some entries are missing due to hardware limitations.

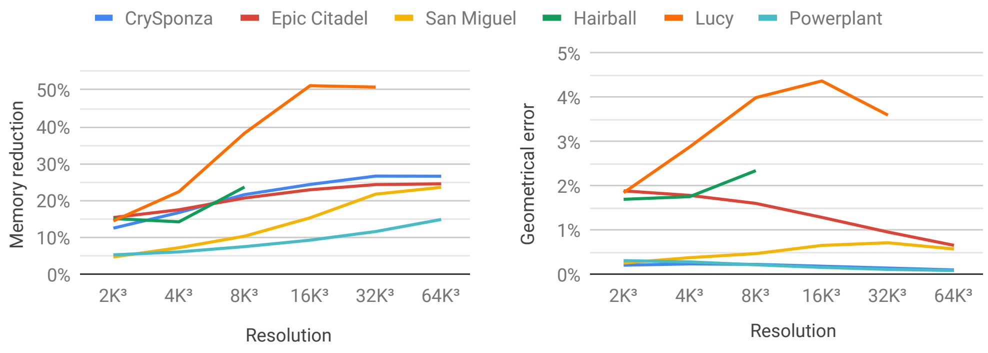


Fig. 10. Left: Memory reduction of the LSVDAG compared to the SVDAG. Right: Error rate of the LSVDAG in percentage of changed voxels out of all non-empty voxels.

in Table 1, where the amount of edges in the similarity graph is roughly three times the amount of nodes, whereas for the Lucy scene there is a twenty to one ratio. The rest of the scenes contain a mix of axis-aligned surfaces (e.g. floor and walls), and randomly oriented surfaces, and thus achieve compression rates between the two previous examples. It is interesting to note that our algorithm is able to leverage the most compression when surface alignment is well distributed, which is precisely where the original SVDAG struggles the most, making it an ideal addition when processing voxel data.

Additionally, the output of our algorithm can also be used as input to other state-of-the-art optimizations of the original SVDAG algorithm, namely pointer compression dubbed as the ESVDAG [Dado et al. 2016; Villanueva et al. 2016] and the Symmetry-aware SVDAG (SSVDAG) [Villanueva et al. 2016]. Fig. 11 showcases the memory ratio of the SVDAG, ESVDAG and SSVDAG relative to their lossy variants, which we name LESVDAG and LSSVDAG respectively, for three of our test scenes. Results for the remaining scenes follow the same trend and can be seen in the

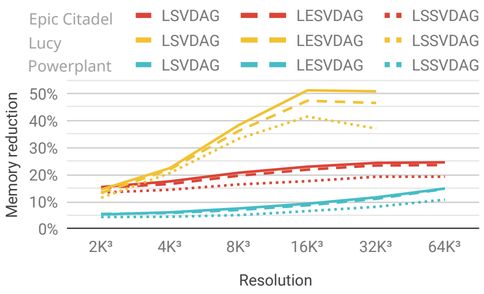


Fig. 11. Memory reduction comparison of the LSVDAG relative to the SVDAG, the LESVDAG relative to the ESVDAG and LSSVDAG relative to the SSVDAG.

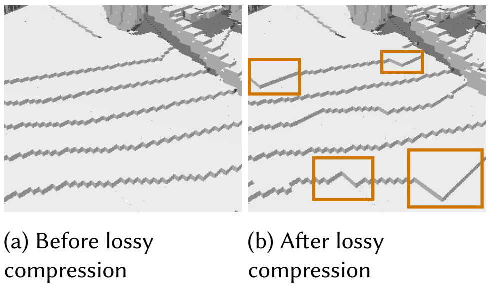


Fig. 12. Left: Original curved surface in the Epic Citadel scene at a resolution of $32K^3$. Right: Square compression artifacts highlighted when using aggressive lossy compression parameters.

supplementary material. All variants of the SVDAG structure achieve a higher compression rate by using our technique. The compression rate of the LESVDAG relative to that of the LSVDAG results in a compression gain roughly 2 to 5% smaller than the gain obtained when applying it on the conventional SVDAG. We theorize this difference is caused by the reduction in the amount of nodes inducing a greater reduction in the amount of pointers, which in turn reduces the maximum potential that the pointer compression can achieve. SSVDAGs benefit less from our lossy compression algorithm compared to conventional SVDAGs (roughly 10% less relative compression), since our method reduces the amount of unique subtrees, which lowers the probability that symmetrically identical subtrees can be found afterwards.

As seen in Fig. 1 and Fig. 13, the errors are well distributed in the scene, and appear mostly in non-planar surfaces, making them hard to notice. Largely planar areas, where errors would be most visible are usually represented by subtrees of high effective reference count, and thus remain unaffected by our compression technique. Due to the cubical shape of the volumes that the nodes represent, artifacts can become slightly more noticeable where curved edges are turned into a square ones. Nevertheless, the geometric deviation is small in this case. An example of this type of artifact is shown in Figure 12. If these artifacts need to be prevented, the similarity measure could be changed to take not only the current subtree but also its immediate neighborhood into account.

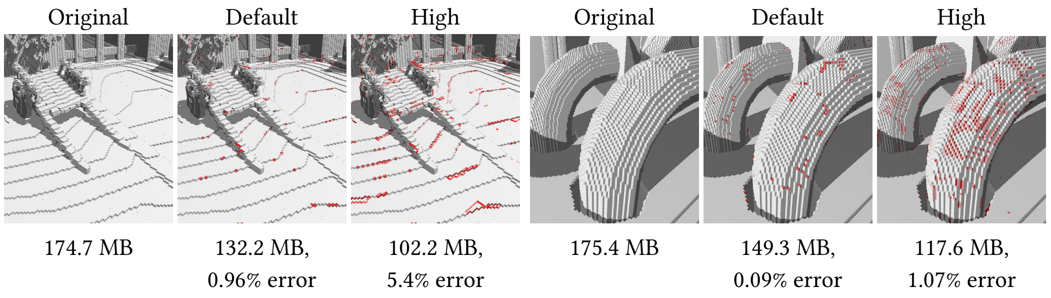


Fig. 13. A visualization of the error introduced by our compression method when using default parameters and high compression parameters (inflation of 1.5, difference factor of 4, and reference count threshold of 4) for the Epic Citadel (left) at $32K^3$ resolution and Powerplant (right) at $64K^3$ resolution. The voxel difference to the original is highlighted in red.

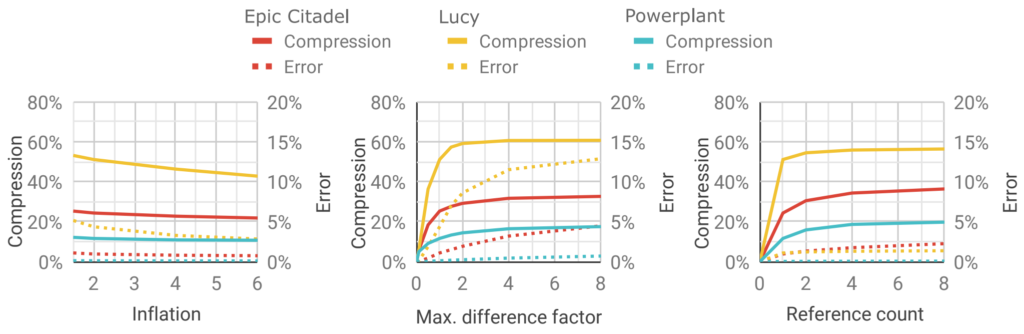


Fig. 14. The influence of changing the lossy compression parameters. In each graph, one of the parameters is changed while the others are kept at default. The percentages on the Y-axis indicates the amount of compression rate in terms of memory consumption and error rate in terms of percentual voxel difference. The data of these graphs was obtained from the Epic Citadel and and Powerplant scenes at a resolution of $32K^3$, and Lucy at a resolution of $16K^3$.

4.3 Parameter evaluation

Fig. 14 shows the influence of the three parameters described in Sec. 3 on the compression ratio and loss for three of our test scenes at a $32K^3$ resolution. The clustering inflation parameter has the smallest relative effect on the final result. Higher values lead to smaller clusters, which reduces the amount of subtrees replaced by a single representative. The threshold on allowed voxel differences has the largest effect on compression and error rate. Allowing much larger differences has a limited effect because we constrain equal topology except for the last level. Finally, including subtrees with larger effective reference counts also quickly shows a flattening of the achieved compression rates, since only a small percentage of subtrees (roughly 5%) are referenced more than three times in the SVDAG. Fig. 13 shows an example of the types of errors introduced by the default and more aggressive compression parameters on selected views of our test scenes.

A possible line of future work is to consider different similarity measures, e.g., a perceptual measure [Nader et al. 2016], or application-driven choices. Furthermore, we employ an off-the-shelf clustering algorithm. A domain-aware clustering algorithm may be able to achieve better results by finding larger subtree groups to cluster. In our current approach, the clustering of nodes is performed separately at every individual level, but it could be useful to explore multi-level clustering. Finally, to make the similarity graph creation tractable for large SVDAGs, we enforce equal topology up to the last level. A higher amount of levels could be used to obtain more matches, with an increase in computation time.

5 CONCLUSION

We have shown that the memory consumption of the SVDAG can be significantly reduced at the cost of only a small loss in precision. Our lossy compression technique clusters similar infrequently referenced subtrees together in order to reduce the resulting error. Our test scenes were compressed on average by 27.4% using default parameters at high resolutions, while usually affecting less than 1% of the original voxels. Additionally, the error is well distributed, making artifacts less noticeable. For more aggressive compression parameters, the amount of compression can increase by an additional 5% to 15%. Our approach can also be further compressed when using optimized SVDAG encodings, such as ESVDAG [Dado et al. 2016; Villanueva et al. 2016] and SSVDAG [Villanueva et al. 2016].

ACKNOWLEDGMENTS

We would like to thank Villanueva et al. for making their source code publicly available, which was used as a foundation for this work [Villanueva et al. 2016]. The Crytek Sponza, San Miguel and Hairball scenes were obtained from Morgan McGuire's Computer Graphics archive. The remaining scenes are courtesy of Epic Games (Epic Citadel), the Stanford 3D Scanning Repository (Lucy) and the University of North Carolina at Chapel Hill (Powerplant). This work was funded by the NWO Vernieuwingsimpuls VIDI grant NextView.

REFERENCES

Chandrajit Bajaj, Insung Ihm, and Sanghun Park. 2001. Visualization-specific compression of large volume data. In *Proceedings Ninth Pacific Conference on Computer Graphics and Applications. Pacific Graphics 2001.* IEEE, 212–222.

Rafael Ballester-Ripoll, Peter Lindstrom, and Renato Pajarola. 2019. TTHRESH: Tensor Compression for Multidimensional Visual Data. *IEEE transactions on visualization and computer graphics* (2019).

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

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

Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games.* ACM, 15–22.

Cyril Crassin, Fabrice Neyret, Miguel Sainz, and Elmar Eisemann. 2010. *Gpu pro.* AK Peters, inbook X.3 Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels, 643–676. http://graphics.tudelft.nl/Publications-new/2010/CNSE10a

Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. 2011. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 1921–1930.

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

Dan Dolonius, Erik Sintorn, Viktor Kampe, and Ulf Assarsson. 2017. Compressing Color Data for Voxelized Surface Geometry. *IEEE transactions on visualization and computer graphics* (2017).

Stijn Dongen. 2000. A cluster algorithm for graphs. (2000).

Enrico Gobbetti, Fabio Marton, and José Antonio Iglesias Guitián. 2008. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer* 24, 7-9 (2008), 797–806.

Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Straßer. 2002. Interactive rendering of large volume data sets. In *Proceedings of the Conference on Visualization'02.* IEEE Computer Society, 53–60.

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

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

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

Timothy R. Kol, Pablo Bauszat, Sungkil Lee, and Elmar Eisemann. 2019. MegaViews: Scalable Many-View Rendering With Concurrent Scene-View Hierarchy Traversal. *Computer Graphics Forum* 38, 1 (2019), 235–247.

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

G. Nader, K. Wang, F. Hétroy-Wheeler, and f. Dupont. 2016. Visual Contrast Sensitivity and Discrimination for 3D Meshes and their Applications. *Computer Graphics Forum* 35, 7 (2016), 497–506.

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

Ruwen Schnabel and Reinhard Klein. 2006. Octree-based Point-Cloud Compression. *Spbg* 6 (2006), 111–120.

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

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

Brent Robert Williams. 2015. Moxel DAGs: Connecting material information to high resolution sparse voxel DAGs. (2015).