TU Delft

Delft University of Technology

Transform-Aware Sparse Voxel Directed Acyclic Graphs

Molenaar, M.L.; Eisemann, E.

DOI
10.1145/3728301

Publication date
2025

Document Version
Final published version

Published in
Proceedings of the ACM on Computer Graphics and Interactive Techniques

Citation (APA)
Molenaar, M. L., & Eisemann, E. (2025). Transform-Aware Sparse Voxel Directed Acyclic Graphs. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, *8*(1). https://doi.org/10.1145/3728301

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Transform-Aware Sparse Voxel Directed Acyclic Graphs

MATHIJS MOLENAAR, Delft University of Technology, Netherlands
ELMAR EISEMANN, Delft University of Technology, Netherlands

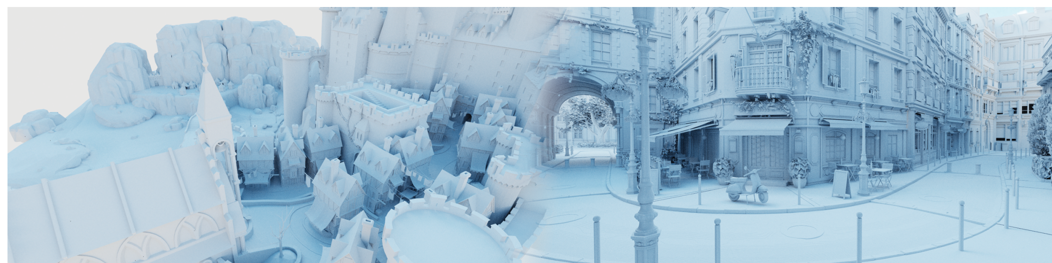


Fig. 1. Two scenes at a $64k^3$ voxel resolution, losslessly compressed using our **T**SVDAG method to 597MB and 721MB, respectively.

Sparse Voxel Directed Acyclic Graphs (SVDAGs) have proven to be an efficient data structure for storing sparse binary voxel scenes. The SVDAG exploits repeating geometric patterns; which can be improved when considering mirror symmetries. We extend the previous work by providing a generalized framework to efficiently involve additional types of transformations and propose a novel translation matching for even more geometry reuse. Our new data structure is stored using a novel pointer encoding scheme to achieve a practical reduction in memory usage.

CCS Concepts: • **Computing methodologies** → Ray tracing; **Volumetric models**.

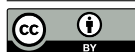
Additional Key Words and Phrases: voxel, sparse voxel directed acyclic graph, SVDAG, sparse voxel octree, SVO, compression, ray tracing

ACM Reference Format:
Mathijs Molenaar and Elmar Eisemann. 2025. Transform-Aware Sparse Voxel Directed Acyclic Graphs. *Proc. ACM Comput. Graph. Interact. Tech.* 8, 1, Article 11 (May 2025), 16 pages. https://doi.org/10.1145/3728301

1 Introduction

The demand for large-scale 3D voxel structures (Figure 1) is growing across various domains, such as 3D reconstruction [Chen et al. 2013], simulation [Hoetzlein 2016; Museth 2013], VR painting [Kim et al. 2018], path guiding [Müller et al. 2017], and real-time indirect lighting [Crassin et al. 2011]. Various solutions have been proposed, tailored for their specific use cases. This eventually leads to trade-offs in the design goals. For example, a representation that is easy to manipulate cannot afford the cost of the most compact compression logic [Museth 2013]. Conversely, the most compressed representations may not support random voxel accesses or efficient rendering algorithms [Ballester-Ripoll et al. 2019].

Authors' Contact Information: Mathijs Molenaar, Delft University of Technology, Delft, Netherlands, m.l.molenaar@tudelft.nl; Elmar Eisemann, Delft University of Technology, Delft, Netherlands, e.eisemann@tudelft.nl.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2577-6193/2025/5-ART11
https://doi.org/10.1145/3728301

In this paper, we focus on improving a specific class of 3D voxel structures, namely *static* Sparse Voxel Directed Acyclic Graphs (SVDAGs). This representation, first suggested in a 2D image context [Parker and Udeshi 2003] and later ported to 3D [Kämpe et al. 2013], extends the concept of Sparse Voxel Octrees by removing duplicate subtrees. The initial work on SVDAGs only considers two subtrees equivalent if they represent the exact same geometry. Later work extends this concept to equivalence under mirror symmetry [Villanueva et al. 2016], and approximate similarity [van der Laan et al. 2020]. We introduce a more general framework for efficiently finding any transformation, which can be described by reordering child nodes; e.g. 90 degree rotations. Additionally, we show how similarity under geometric translation further decreases the size of the structure. Finally, we propose a new compact pointer scheme that aids in reducing memory usage.

2 Previous Work

Volumetric data is used in many fields both inside and outside of computer graphics, including rendering [Gobbetti et al. 2012; Guthe and Goesele 2016; Yeo and Liu 1995], simulation [Museth 2013], VR painting [Kim et al. 2018], and 3D reconstruction [Chen et al. 2013]. Each use case poses a different set of requirements, leading to a large variety of data structures. We focus on voxels, defined on a regular grid; other volumetric-data types will not be discussed.

2.1 Decomposition-Based Methods

While dense grids are fast and easy to implement, their memory requirements prohibit their usage for high-resolution data sets. Due to the need for high compression ratios, most techniques offer a lowered quality to reduce memory usage. This is typically achieved by subdividing the data set into smaller "bricks" of N^3 voxels, where N is a small number such as 4 or 8. These bricks are compressed by describing them as a weighted combination of 3D basis functions and quantizing the corresponding coefficients.

Gobbetti et al. [2012] efficiently construct a dictionary of basis functions using *coresets* and *K-SVD* [Aharon et al. 2006]. Yeo and Liu [1995] apply the JPEG compression scheme to three-dimensional data, using the Discrete Cosine Transform (DCT) to create basis functions. Muraki [1993] uses wavelets to decompose a dense grid. TThresh [Ballester-Ripoll et al. 2019] uses a Tucker-based decomposition followed by various compression steps.

2.2 Sparse Volumes

Decomposition-based methods have been shown to achieve high compression ratios on dense scientific datasets, such as medical imaging. However, there are also many other use cases, specifically in real-time applications, where volumetric data consist mostly of empty space. Examples include 3D texturing [Benson and Davis 2002; DeBry et al. 2002], VR painting [Kim et al. 2018], and 3D reconstruction [Chen et al. 2013]. Here, fast display and/or editing capabilities are required.

The spatial hash [Gaede and Günther 1998] stores the voxels in a hash table using a 3D hash function. Although the amortized search time is $O(1)$, the worst case is $O(N)$. *Perfect* spatial hashing [Lefebvre and Hoppe 2006] guarantees constant-time accesses using a perfect hash function. More recently, spatial hashing has been used to accelerate real-time light transport [Gautron 2022].

OpenVDB [Museth 2013] combines spatial hashing with a shallow N-ary tree to provide good data locality. Memory is saved by omitting empty regions from the tree. This work was later modified to run efficiently on the GPU [Hoetzlein 2016; Museth 2021]. Recently, Kim et al. [2024] achieved significant memory savings by replacing the lower part of the tree with a neural representation. Crassin et al. [2009] use N-trees to build an LOD-based out-of-core GPU renderer, including a scene graph [Crassin et al. 2010]. Laine and Karras [2010] proposes an extension to octrees to more accurately represent 3D surfaces.

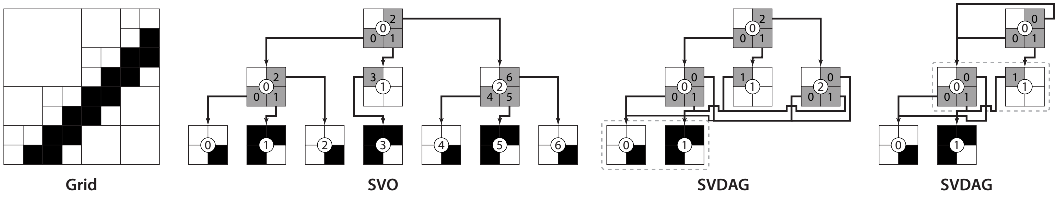


Fig. 2. A two-dimensional illustration how a Sparse Voxel Octree (left) can efficiently be converted to a Sparse Voxel Directed Acyclic Graph (right) from the bottom-up. At each step, all duplicate nodes/leaves from a single level are removed by only considering the child pointers.

2.3 Sparse Voxel Directed Acyclic Graphs

Voxel Octrees are trees organizing three-dimensional data with a branching factor of $N = 2$ along each dimension, that is: the grid is recursively subdivided into equally sized (and non-overlapping) regions. For sparse data sets, *Sparse* Voxel Octrees (SVOs) save memory by not subdividing empty regions further, which also accelerates ray tracing through empty-space skipping.

Sparse Voxel *Directed Acyclic Graphs* (SVDAGs) [Kämpe et al. 2013; Parker and Udeshi 2003] aim to reduce the number of nodes inside a Sparse Voxel *Octree*, by finding and removing identical subtrees. This creates subtrees with multiple parent nodes, turning the tree into a *Directed Acyclic Graph*. The search for duplicate subtrees is very efficient and enables real-time edits [Careil et al. 2020; Molenaar and Eisemann 2024], albeit on a limited scale. SVDAGs have also been used for compact storage of shadow maps [Scandolo et al. 2016], shadow volumes [Sintorn et al. 2014], and animated volumes [Kämpe et al. 2016].

The key for SVDAG compression to work is that the data set must contain many equivalent subtrees. Encoding attribute data, such as density or colors, directly into the voxels significantly reduces this property, as subtrees will only be equivalent if they match in attributes and geometry. Instead, various works [Dado et al. 2016; Dolonius et al. 2017; Molenaar and Eisemann 2023; Williams 2015] suggest separating geometry from attributes. The attributes of the non-empty voxels are collected along a Morton space filling curve [Morton 1966] and compressed separately. Thus, the SVDAG stores binary occupancy, and to index the attribute array, it stores additional data inside nodes [Dado et al. 2016] or child pointers [Dolonius et al. 2017].

Two additional methods have been proposed to further increase the reuse of subtrees within the SVDAG. Lossy compression [van der Laan et al. 2020] matches similar (not necessarily identical) subtrees using a user-defined quality threshold. Villanueva et al. [2016] propose a method to efficiently find and encode subtrees that are equivalent under mirror symmetry, dubbed Symmetry-Aware Sparse Voxel Directed Acyclic Graphs (**S**SVDAG).

3 Background

In this section, we provide the necessary background for readers unfamiliar with (Symmetry-Aware) Sparse Voxel Directed Acyclic Graphs (**S**SVDAGs), and we present an improvement to the **S**SVDAG construction algorithm. We will start with a brief explanation of Sparse Voxel Octrees (SVOs), followed by how they can be turned into SVDAGs and **S**SVDAGs. We will only consider binary voxel data.

3.1 Sparse Voxel Octree (SVO)

A Voxel Octree is a tree over a voxel grid. Starting with a single voxel representing the entire scene, which is set to 1 if it contains scene geometry (occupied) and 0 otherwise (empty). It is then

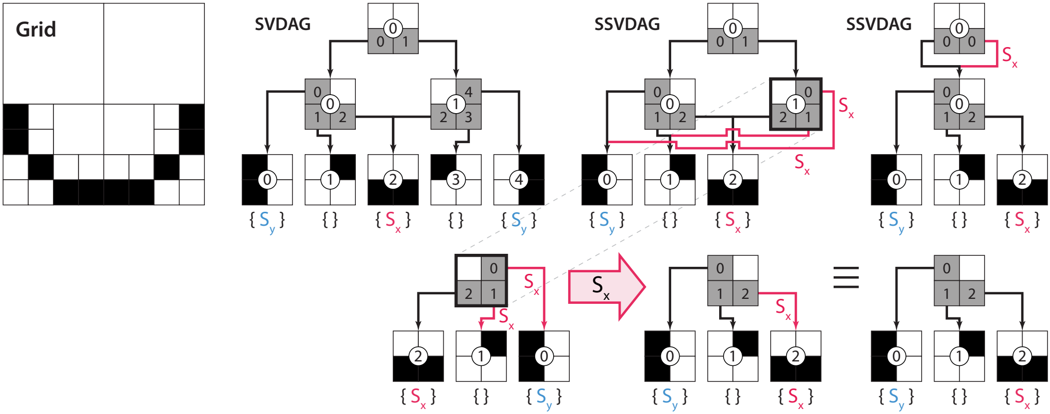


Fig. 3. Top row: converting an SVDAG to a Symmetry-Aware SVDAG (**S**SVDAG). Underneath each leaf we denote the symmetries to which it is invariant. Bottom row: after a mirror transformation, the two inner nodes are equivalent. One child received a different x transformation (pink) but is invariant to it.

subdivided into octants that are recursively treated. This process repeats for L steps. The resulting voxel hierarchy levels are enumerated from the root level (L) to leaf level (0). In practice, most implementations stop the recursion at level 2 and store the remaining subtree as a dense 4^3 voxel grid. A *Sparse* Voxel Octree omits octants that do not contain any occupied voxels, reducing the number of tree nodes in sparse data sets (Figure 2).

3.2 Sparse Voxel Directed Acyclic Graph (SVDAG)

At high voxel grid resolutions, duplicate subtrees (thus repeating geometry) are common. Rather than storing all subtrees, a Sparse Voxel *Directed Acyclic Graph* (SVDAG) stores only a single subtree representative.

SVDAGs are typically constructed from an existing SVO. This conversion process can be implemented very efficiently using a bottom-up algorithm [Kämpe et al. 2013; Parker and Udeshi 2003]. Starting at level $L = 1$ (where $L = 0$ stores individual voxels), we can trivially find and fuse duplicates by flattening the subtrees into dense 2^3 voxel grids. Moving up to the parent level $L = 2$, we now know that children representing the same geometry must have the same pointers. This allows us to find duplicate subtrees in $L = 2$ *only* by looking for nodes that have the exact same child pointers, without any further recursion (Figure 2). This process is repeated for all subsequent levels until the root node is reached.

3.3 Symmetry-Aware SVDAG (SSVDAG)

Symmetry-Aware SVDAGs [Villanueva et al. 2016] (**S**SVDAGs) have been introduced as the first and, so far, only method to support transformations inside an SVDAG. The core idea is that two subtrees can be duplicates if their geometry matches after mirroring around the primary axes (Figure 5). In this scheme, the child pointers encode both the memory addresses and the mirror axes. The latter are encoded using 3 bits, one bit per axis (x, y, z).

Leaves. Construction of an **S**SVDAG follows a structure similar to that of a regular SVDAG. The subtrees at the level $L = 1$ are flattened into 2^3 voxel grids. Finding duplicates with respect to symmetry transformations is more complicated than a regular SVDAG, as there may be multiple

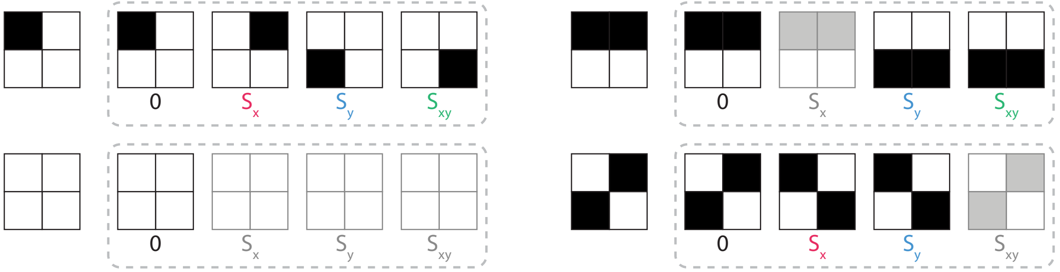


Fig. 4. Different 2^2 leaves (left) that can be transformed into the same *canonical representation* (right) in **S**SVDAG. Grids displayed in gray indicate invariance to that transformation.

different grids g_i, which become equivalent after a symmetry transformation $S_{x,y,z}(g)$ is applied. We consider two grids equivalent when they represent the exact same geometry (Figure 4). We denote that using ≡. This raises the question of which grid g_i should be chosen as the representative. The suggested solution[Villanueva et al. 2016] is to define, for each 2^3 grid g, the *canonical representation* $g^\star = \text{argmax}_{S_{x,y,z}} B(S_{x,y,z}(g))$, where $B(g)$ is the binary number when concatenating all voxels of g. This *canonical representation*, and the accompanying transformation, are precomputed for each possible 2^3 voxel grid and stored in a look-up table. Duplicates can then be easily detected by comparing the *canonical representation* $g^\star$ of each grid g.

Inner Nodes. To search for duplicate inner nodes ($L > 2$), Villanueva et al. [2016] recognize that mirror symmetry can be applied recursively. To apply a symmetry transformation to an inner node, one swaps its children along the respective symmetry axis and then applies the same transformation to those subtrees (Figure 3). By processing the tree from the bottom up, the child subtrees c_i, c_j that are equivalent under *one* symmetry transformation (i.e. $c_i \equiv S_{x,y,z}(c_j)$), are guaranteed to have the same child pointer addresses. To determine whether two subtrees are equivalent under a *specific* transformation $S_{x,y,z}$, each subtree maintains a (temporary) 3-bit mask, storing its invariance to x, y, or z symmetry ($g \equiv S_{x,y,z}(g)$). To search for duplicate nodes, the search space is first reduced by clustering nodes with the same child pointers. An exhaustive search for the eight possible transformations $S_{x,y,z}$ is used to find all representative inner nodes and compute their respective mirror invariance.

3.4 Improving Symmetry-Aware SVDAG

In the original work [Villanueva et al. 2016], the authors assume that the invariance to symmetry along the x, y, or z axes is independent, and thus a 3-bit mask $M = \{m_x, m_y, m_z\}$ is sufficient to encode all invariances. However, while an invariance along, for example, the x and y axes implies xy invariance, the opposite does not always hold. The right/bottom example given in Figure 4 illustrates a grid invariant under xy, but not under x or y symmetry. This type of invariance is currently missed by the original **S**SVDAG method [Villanueva et al. 2016], which leads to an increase in the number of inner nodes. This issue can be easily remedied using a 7-bit mask to store all invariance $(x, y, z, xy, yz, xz, xyz)$. We found that this has an impact on the compression ratio of **S**SVDAG (Table 2).

4 Our Method

We extend the concept of **S**SVDAGs (Section 3.3) to support all transformations that can be described by reordering the children of the nodes to remove geometric redundancy. We then select

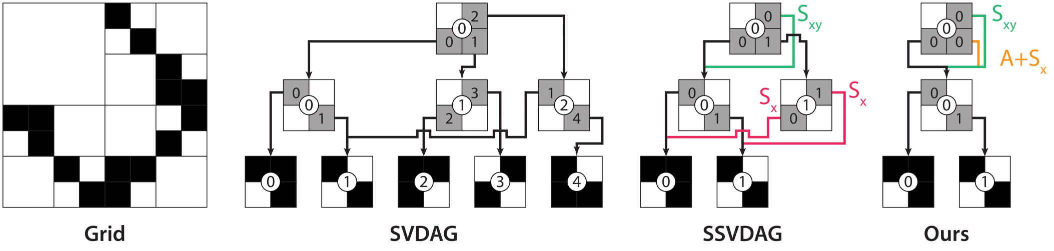


Fig. 5. Example of a 2D grid represented by different data structures. The SVDAG contains 9 items. This is reduced to 5 items by the (improved) **S**SVDAG [Villanueva et al. 2016]. Our method represents the same grid with just 4 items. S_i indicates mirror symmetry along axis i; A indicates that the x and y axes are swapped.

a suitable subset of transformations to achieve practical compression. In addition, we introduce a new matching via geometric translations. In Section 5, we will describe a method for efficient encoding.

4.1 Arbitrary Transformations

We define a node N as a tuple of eight child pointers, one for each octant: $N := (c_0, \ldots, c_7)$. A child pointer consists of a memory address and a transformation that is applied to that child; empty octants are represented by a null pointer ($c_i = \varnothing$). We will first consider all 8! permutation transformations that can be applied to the set of child octants. We define a permutation as a bijective function $T : \{0, \ldots, 7\} \longrightarrow \{0, \ldots, 7\}$. To obtain a transformed child pointer $T(c_i)$, we concatenate T with the transform stored in c_i. Like Symmetry-Aware SVDAGs (**S**SVDAGs), these permutations are applied recursively to all child levels. A transformed node $T(N)$ consists of the shuffled children with the transform T applied to each child pointer. In theory, it would be possible to select a variable subset of children to which this transformation is applied, but for the transformation subset that we chose to use in practice, it proved beneficial to apply it to all children. However, more exploration would be possible.

Leaves. Like in previous work, we flatten subtrees at level $L = 1$ into 2^3 voxel grids g_i. Rather than searching only for exact geometric matches, we need to consider that two *different* voxel grids can be deduced from the *same* representative grid using different transformations. As such, we need a deterministic method to pick a single representative. Let $\hat{G}$ be the set of grids that can be transformed into g: $\hat{G} = \{\hat{g} \mid T(\hat{g}) \equiv g\}$. We define the *canonical representation* $g^\star$ of a grid g as the "highest-value" grid in $\hat{G}$: $g^\star = \text{argmax}_{\hat{g} \in \hat{G}} B(\hat{g})$, where $B(g)$ is the binary number obtained by concatenating all voxels of a voxel grid. Note that this definition differs slightly from that of **S**SVDAG (Section 3.3), which assumed that $T^{-1} = T$, which is not generally the case.

Using a precomputed look-up table, we replace each leaf by a *canonical-transform pointer* (containing the canonical grid $g^\star$ and the corresponding transformation) and a (temporary) bitmask that stores the invariance of $g^\star$ with respect to all possible transformations ($g^\star \equiv T(g^\star)$). This mask requires $8! = 40,320$ bits, which limits the practical scene size. We will address this later by reducing the number of permutations that are considered.

Inner Nodes. When processing level L, we know that two subtrees at level $L - 1$ are equivalent under *some* transformation if and only if they share the same *canonical representation* (canonical-transform pointer address). To reduce the search space, we first create clusters of nodes that have the same child pointer addresses. Note that the order in which the children appear or the transformations that are applied to them may be different. We then iterate through the nodes

Table 1. The number of nodes & 4^3 leaves when considering only the mirror symmetry (S) and axis permutation (A) subset $S + A$, rather than all possible permutations. All scenes use a resolution of $16k^3$.

Scene	SVDAG	S	A	S+A	All
Bistro	6.74M (+41.3%)	5.59M (+17.1%)	6.08M (+27.5%)	4.94M (+3.6%)	4.77M (100%)
Bunny	9.75M (+103.1%)	6.44M (+34.2%)	6.89M (+43.6%)	4.81M (+0.2%)	4.80M (100%)
Crown	23.01M (+65.5%)	16.88M (+21.4%)	18.03M (+29.6%)	14.00M (+0.6%)	13.91M (100%)
Citadel	9.41M (+67.9%)	6.89M (+23.0%)	7.56M (+35.0%)	5.64M (+0.7%)	5.60M (100%)
Lucy	5.68M (+45.5%)	4.71M (+20.6%)	4.92M (+26.0%)	3.92M (+0.6%)	3.90M (100%)
Dragon	5.97M (+45.3%)	4.96M (+20.8%)	5.12M (+24.8%)	4.13M (+0.6%)	4.11M (100%)

in each cluster, testing for each node N_i whether it can be deduced from any of the previous nodes: $\exists j < i, T : N_i \equiv T(N_j)$. We perform this test using an exhaustive search for all possible transformations T. If no match is found, then we consider the node a *canonical representation*, computing its invariance and outputting the node.

To test whether $N_i \equiv T(N_j)$, we first apply the permutation T to node N_j, which reorders the children and updates the transformations stored in their pointers. Two transformed child subtrees are equivalent $T_i(c_k) \equiv T_j(c_k)$ if they have the same *canonical representation* c_k (canonical-transform pointer address) and their transformations result in the same geometry. The latter can be tested by checking the invariance bit mask of the *canonical representation* for transformation: $c_k \equiv T_i^{-1} \circ T_j(c_k)$.

Practical Subset. Using *all* possible transformations reduces the number of tree nodes the most, but is impractical when it comes to encoding these transformations. We aim to find a small subset of transformations that still cover most reuse. Any valid subset must adhere to two requirements. First, the combination of two transformations in the subset must also be in the subset: $\forall T_i, T_j \in \Omega : T_i \circ T_j \in \Omega$. Second, for any transformation in the subset, the subset must also contain the inverse: $\forall T_i \in \Omega : T_i^{-1} \in \Omega$.

In practice, we found that the combination of two types of geometric transformations together accounts for almost all reuse (Table 1). Symmetries flip children along the primary axis; this is the same as **S**SVDAG [Villanueva et al. 2016] (with the improvement discussed in Section 3.4). Axis permutations swap the primary axis (e.g. $(x, y, z) \longrightarrow (z, x, y)$); in total, there are 6 different axis permutations. When combined, these two types of transformation can represent any 90-degree rotation along one of the primary axes, which was not previously possible [Villanueva et al. 2016]. This is the symmetry group of the cube and the octahedron.

4.2 Translations

A general limitation of SVDAGs is that repeating geometry presents itself only as duplicate subtrees if they align with the SVDAG node boundaries. This can be partially remedied by including geometric translation. More specifically, we consider what we name *destructive* translations. This means that, when translating voxels of a $(2^L)^3$ subtree, any voxels that are translated "out" of the subtree will disappear (Figure 6).

The search space of all possible translations becomes very large for deep subtrees. We therefore only consider translating one subtree to obtain another, instead of applying destructive translations on both, and limit ourselves to searching only for translation matches in subtrees of up to 16^3 voxels ($L = 4$). In practice, higher levels have diminishing returns anyway, as the increased translation range $[-2^L + 1, +2^L - 1]^3$ takes more bits to encode.

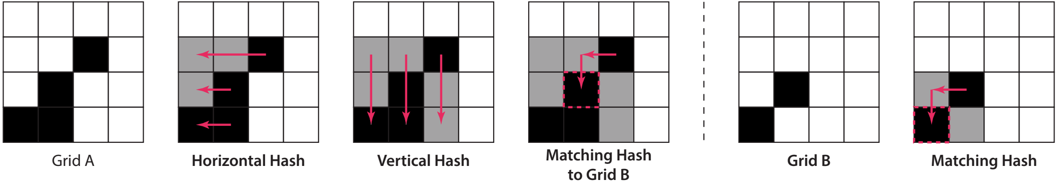


Fig. 6. We compute for each voxel a hash describing the area to the top/right. The hash is first computed horizontally, and then vertically. The hash at the lower/left voxel of grid B matches a voxel in grid A, indicating that B could be constructable from A by a translation of $(-1, -1)$.

Because translations operate across the boundaries between child subtrees, it is difficult to describe them in terms of SVDAG node/leaf operations. The subtrees are therefore flattened into $(2^L)^3$ voxel grids. We initially experimented with a naive search, iterating over the N voxel grids, applying each possible translation, and subsequently searching for a matching grid (using a hash table). This naive search has a time complexity of $O(NV^2)$, where $V = (2^L)^3$ is the number of voxels. The quadratic term originates from the fact that there are $O(V)$ different translations, which take $O(V)$ to apply. Although we consider SVDAG compression to be a preprocess; $O(NV^2)$ is impractical (weeks of computation for $64K^3$ scenes on a modern consumer computer).

Fast Translation Search. We propose a faster matching solution to reduce the complexity to $O(NV)$. We solve the problem for groups of translations at a time, distinguished by the direction of the translations along each axis. For example, in the following, we will consider translations of the form $(-x, -y, -z)$ with $x, y, z \geq 0$. The process for all seven other direction groups is similar.

For each voxel, we compute a hash value that summarizes the voxel grid in the $(+x, +y, +z)$ direction; assuming that all voxels beyond the subtree boundary are empty. We then find possible matches via translation in the $(-x, -y, -z)$ direction. If the hash of a voxel at the most extreme subtree location in direction $(-x, -y, -z)$ matches any of the other voxel hashes in another grid, we have found a potential match (e.g., if (i_x, i_y, i_z) is a matching voxel, its translated subtree by $(-i_x, -i_y, -i_z)$ is a potential match). Since two different grids may produce the same hash code (hash collision), we confirm potential matches by translating the grids and testing for equivalence.

The hash computation over a running window (a *rolling hash*) can be accelerated [Karp and Rabin 1987; Rabin 1981]. However, rolling hashes are typically not suitable for input with a small alphabet size (just 0 and 1 in our case), resulting in a high number of hash collisions. Instead, we reformulate our problem so that a regular hash function can be used.

Consider a one-dimensional example: rather than computing a hash value based on the full region to the right of each voxel, we compute a hash that describes the region to the rightmost *occupied* voxel. The hash values can now be computed with a linear scan, starting with the rightmost occupied voxel: $H(v_i) = \text{hash_combine}(H(v_{i+1}), v_i)$; we use an implementation of the Boost C++ library [Boost 2024]. This method can be easily extended to our 3D grids. Similarly to separable image filters, we apply the hash function along each of the primary axes, one at a time (Figure 6). With this solution, finding all translations for the $L = 4$ (16^3) nodes takes a couple of hours for a $64K^3$ scene on the CPU. Note that this is significantly slower than the hierarchical search for mirror symmetry and axis permutations, which can typically be performed in less than a minute.

We can test the full set of our proposed transformations in a unified way. While testing for translations, we can check matching between different directions to additionally search for symmetry as well. For example, a voxel in direction $(+x, +y, -z)$ that matches a corner voxel (of a different grid)

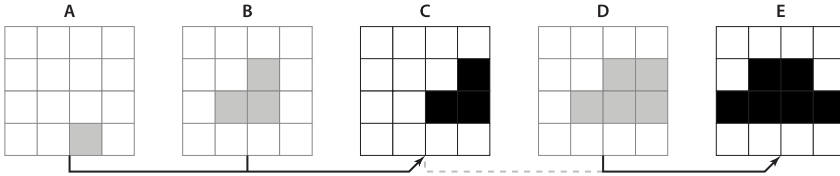


Fig. 7. Five grids and their simplified dependency graph, storing only a single match per grid. **E** is stored as a unique subtree because it cannot be constructed from any of the other grids. **C** must be stored because **A** and **B** depend on it. The remaining grids (gray) can be constructed from **C** and **E**.

in direction $(+x, +y, +z)$, indicates that one can be transformed into the other by a combination of translation and z-axis symmetry. We use a similar process to search for permutations of axes.

Dependency Graph. A consequence of allowing *destructive* translations is that the dependency graph between transformed subtrees becomes complex. To reduce memory usage during compression, we only store a single "parent" per subtree, rather than the entire dependency graph. A parent is another subtree that can be transformed into the current subtree. If there are multiple parents, then we store the one with the highest memory address. This encourages clusters of similar subtrees to pick the same parent. We sort the subtrees in advance based on their number of occupied voxels to favor the most destructive translations. The resulting reduced dependency graph may contain multiple levels of indirection (Figure 7). That is, a subtree that can be constructed from another subtree, which itself can be constructed from yet another subtree. Due to limitations in the SVDAG encoding, we are forced to duplicate subtrees in these cases (**C** in Figure 7).

5 Implementation

Achieving a high compression ratio requires not only a reduction in the number of nodes/leaves, but also a compact encoding scheme. Such a scheme should be compact, but still enable direct traversal of the SVDAG. This precludes the use of pointerless encoding, such as those presented in [Kämpe et al. 2016; Madoš and Ádám 2021]. Instead, we use a more traditional encoding where a node is represented by a bitmask to indicate which children are nonempty, followed by child pointers to these children.

5.1 Transform ID

Our **T**SVDAG complicates pointer encoding, as they do not only contain a memory address, but potentially also a child-reorder transformation (symmetry & axis permutation) and a translation. We encode all transformations in the least-significant bits, followed by the corresponding memory address.

Each child reorder transformation in our subset (Section 4.1) is assigned a unique number. For translations, we find that the most common translation distance along any axis is 0, with other distances distributed somewhat uniformly. Thus, we treat a translation of 0 as a special case. We combine the child-reorder transformation identifier with a 3-bit mask indicating a 0 translation along each of the primary axes. This combined identifier, which we call the *Transform ID*, is stored using Huffman coding to ensure that the most frequently occurring transformations use the least number of bits. We limit the length of Huffman codes [Moffat 2019] to prevent very rare transforms from creating extremely long pointers. The *Transform ID* is stored in the least significant bits of the pointer, followed by a translation distance for each axis with a non-zero translation, and finally the memory address to which it is pointed (Figure 8).

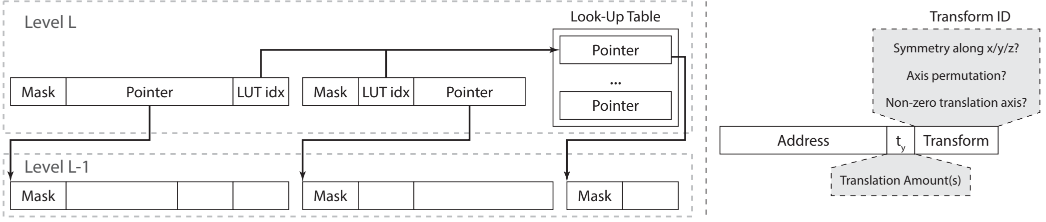


Fig. 8. Left: illustration of two nodes at level L, each with two children. The children are referenced either directly or through a look-up table. Right: example of a pointer with a translation along the y axis. The *Transform ID* encodes symmetry, axis permutation and non-zero translation axis. The translation amounts (t_y) are stored in the more significant bits.

5.2 Node Encoding

Similar to [Laine and Karras 2010; Villanueva et al. 2016], we opt for variable-length pointer encoding, with either 16-, 32-, or 48-bit pointers. The latter is necessary, as encoding transformations create significantly longer pointers. Like Villanueva et al. [2016], we sort our nodes such that the most referenced ones appear first. This ensures that the most frequently used pointers are the shortest.

Pointer Tables. Because of variable-length node encoding, our address space is indexed at 16 bit intervals rather than at node intervals. As a result, many memory indices are never referenced because they do not point to the start of a node. This significantly reduces the number of child pointers that can utilize the compact 16-bit and 32-bit encoding. Second, transformation encoding significantly bloats the pointer sizes. This incurs a heavy penalty if the same pointer occurs many times.

To remedy both issues, we selectively store pointers in a look-up table using fixed size entries of 64 bits (Figure 8). This is somewhat similar to the far pointers of Laine and Karras [2010]. 16-bit pointers now always encode an index into the look-up table, whereas 32-bit pointers reserve 1 bit to indicate whether they contain a table index or a direct pointer; 48-bit pointers are always direct pointers. One look-up table is constructed per SVDAG level using a greedy algorithm. First, the pointers are sorted according to their frequency (how often they occur in the parent level). We then iterate over the sorted pointers, comparing the cost of storing them directly, versus in the look-up table, and subsequently insert them into the look-up table if it is the cheaper option. The cost of a direct pointer is simply its length in bits, rounded up to either 32-bits or 48-bits, times the number of times it occurs. A table pointer incurs a fixed cost of 64 bits per entry, but the resulting 16- or 32-bit pointer will result in lower overall memory usage if it is referenced frequently.

Node Header. The lengths of the child pointers are encoded using a 16-bit mask at the start of the node. Conceptually, this represents eight 2-bit integers storing the length of the child pointers, where a length of 0 represents an empty child. Some tasks, such as rendering, require accessing the nth child pointer. This can be efficiently computed using the *__popc* intrinsic:

Algorithm 1 Efficient extraction of the n'th child pointer.

$$prefix \leftarrow nodes[nodeStart] \; \& \; (0xFFFF >> (16 - 2 * childIndex))$$
$$sum \leftarrow __popc(prefix) + __popc(prefix \; \& \; 0b1010101010101010)$$
$$pointer \leftarrow nodes[nodestart + 1 + sum]$$

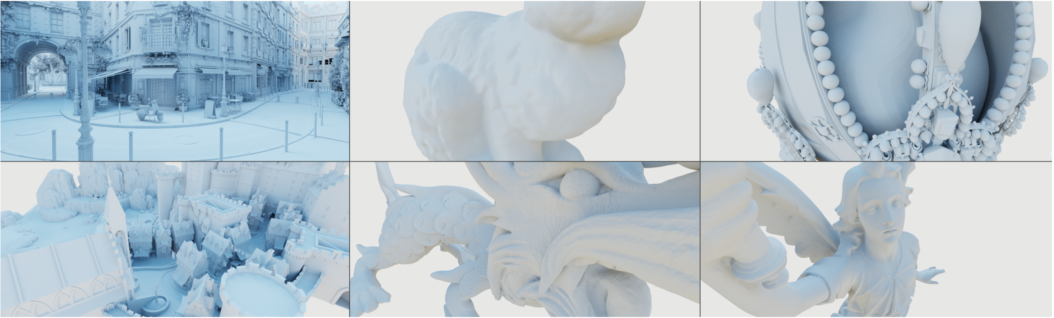


Fig. 9. The scenes used for our evaluation at a voxel resolution of $64k^3$.

5.3 Rendering

To visualize our **T**SVDAGs, we implement a CUDA-based ray tracer that can display various SVDAG formats. Path tracing uses a megakernel approach, although a wavefront implementation is also available [Laine et al. 2013]. We also experimented with replacing the top levels of the SVDAG, typically showing very little reuse, by a hardware-accelerated BVH using Optix. We found that this reduces performance, despite promising findings by Söderlund et al. [2022].

Ray Traversal. Ray intersection is implemented as a stack-based traversal. The intersection points between the ray and the three primary planes passing through the center of a node are computed, from which a bitmask is created indicating which children are intersected. The traversal order is determined by the sign bits of the ray direction following the source code of Careil et al. [2020].

Child Reordering. Our renderer supports a subset of transformations, namely mirror symmetry and axis permutations (Section 4.1). The transformation state is stored on the traversal stack, enabling efficient backtracking. To reduce the size of the stack, we store the transformations by assigning them a unique identifier; this is similar to the *transform ID* (Section 5) but without the translation part. A look-up table in GPU constant memory is used to apply the respective permutations. A similar table stores the result of applying two transformations ($T = T_i \circ T_j$) as defined in these compact identifiers.

Translation. Our definition of *destructive* translation requires careful adjustments to the traversal algorithm. When encountering a translating pointer, the contents of the subtree should be translated; removing any voxels that are moved outside of the bounds of the (non-translated) subtree. This is achieved using a second stack only for those levels of the **T**SVDAG which may contain translations (up to 16^3). Each entry in this stack stores the accumulated amount of translation and the distance at which the ray entered & exited the subtree (tmin & tmax). These values are used to translate the child node centers and to limit their intersection bounds.

6 Results & Discussion

We evaluated our **T**SVDAG method on a variety of different inputs (Figure 9). We chose a combination of architectural scenes (Citadel, Bistro), artist-made models (Crown), and 3D scans of real-world objects from the Stanford 3D Scanning Repository (Bunny, Lucy, Dragon). These were turned into voxel models using the voxelizer in [Villanueva et al. 2016]. For the Citadel scene, we removed most of the surrounding landscape, making the scene more cube-shaped, leading to a higher voxel occupancy. To add context to our results, we include comparisons against a Sparse Voxel Octree, SVDAG, and **S**SVDAG. All structures encode 4^3 leaves using 64 bits.

Table 2. Number of nodes and 4^3 leaves for a Sparse Voxel Octree (SVO), Sparse Voxel Directed Acyclic Graph (SVDAG), Symmetry-Aware Sparse Voxel Directed Acyclic Graph (**S**SVDAG), and our method(s). We present our method with symmetry S, axis permutation A, and translation T.

Scene	SVO	SVDAG	**S**SVDAG	S	A	S + A	S + A + T
Bistro ($64k^3$)	1692.1M	69.6M	56.6M	55.2M	60.0M	47.6M	**41.1M**
21.4B occupied voxels	(+2889%)	(+23%)	(100%)	(-2%)	(+6%)	(-16%)	**(-27%)**
Bunny ($64k^3$)	1267.2M	75.0M	58.0M	55.5M	59.0M	45.0M	**29.2M**
15.2B occupied voxels	(+2085%)	(+29%)	(100%)	(-4%)	(+2%)	(-22%)	**(-50%)**
Crown ($64k^3$)	3205.8M	178.3M	143.4M	137.7M	150.3M	114.5M	**83.5M**
38.6B occupied voxels	(+2136%)	(+24%)	(100%)	(-4%)	(+5%)	(-20%)	**(-42%)**
Citadel ($64k^3$)	1428.8M	65.3M	52.8M	48.7M	54.8M	41.6M	**28.2M**
17.2B occupied voxels	(+2607%)	(+24%)	(100%)	(-8%)	(+4%)	(-21%)	**(-47%)**
Lucy ($64k^3$)	526.6M	52.9M	40.3M	39.1M	41.4M	32.4M	**26.3M**
6.3B occupied voxels	(+1207%)	(+31%)	(100%)	(-3%)	(+3%)	(-20%)	**(-35%)**
Dragon ($64k^3$)	523.9M	53.7M	40.6M	39.5M	41.1M	32.7M	**26.2M**
6.3B occupied voxels	(+1191%)	(+32%)	(100%)	(-3%)	(+1%)	(-19%)	**(-35%)**

6.1 Structure Size

We will first consider the effectiveness of our methods in reducing the size of the structure, measured in the number of nodes & leaves. Table 2 shows the total number of elements (nodes + leaves) for various scenes and methods. The odd numbered rows count the number of elements as a percentage relative to **S**SVDAG [Villanueva et al. 2016]. We abbreviate symmetry, axis permutation, and translation by S, A, T, respectively. Translations are computed for nodes/leaves at the levels representing grids of 16^3, 8^3 and 4^3 resolution.

When comparing **S**SVDAG to our improved method of finding mirror symmetry (Section 3.4), we see that our fix indeed improves the compression ratio. The number of elements marked as duplicates increases by a couple percent depending on the scene. Only permuting the primary axes, without symmetries, leads to a notable reduction in the number of nodes and leaves compared to a regular SVDAG. When combining both, we see a significant improvement over previous work, which only considers axis mirroring. Translating subtrees of resolution 4^3, 8^3, and 16^3 further reduces the number of nodes and leaves. The latter suggests that a higher compression ratio may be achieved when one is not limited to only transformations that can be described as a recursive reordering of child pointers.

6.2 Pointer Compression

Reducing the number of elements is only half of the story to achieve a good compression ratio. In theory, with fully arbitrary transformations, a single canonical representation would suffice to represent all subtrees. However, in practice, the cost of encoding these transformations would outweigh the cost of simply storing duplicates. Thus, we evaluate how the presented encoding scheme (Section 5) allows us to compactly store our **T**SVDAGs with various types of transformations.

Table 3 evaluates the impact of our pointer encoding scheme for our full **T**SVDAG (symmetry, axis permutation, and translation). Without pointer look-up tables ("Base" in Table 3), we find that some scenes fail to encode due to running out of the 48-bit pointer space. This typically happens for pointers at level $L = 4$ with translations along multiple axes, as storing the translation amounts requires 5 bits per axis (in addition to the *Transform ID*). Introducing a look-up table for large- or frequently used pointers solves this issue by providing storage for 64-bit pointers. As expected,

Table 3. The impact of the different features of our pointer encoding on the overall size of the Transform SVDAG. N/A indicates that the scene could not be encoded due to running out of pointer bits.

Scene	Base	Pointer LUT	Pointer LUT + Huffman
Bistro ($64k^3$) 21.4B occupied voxels	876MB (+16%)	753MB (100%)	**721MB** **(-4%)**
Bunny ($64k^3$) 15.2B occupied voxels	N/A	718MB (100%)	**663MB** **(-8%)**
Crown ($64k^3$) 38.6B occupied voxels	N/A	1965MB (100%)	**1851MB** **(-6%)**
Citadel ($64k^3$) 17.2B occupied voxels	715MB (+14%)	625MB (100%)	**597MB** **(-4%)**
Lucy ($64k^3$) 6.3B occupied voxels	683MB (+20%)	567MB (100%)	**535MB** **(-6%)**
Dragon ($64k^3$) 6.3B occupied voxels	681MB (+19%)	571MB (100%)	**540MB** **(-5%)**

the additional indirection helps to reduce memory usage, as frequently used pointers can now be stored as a small index into the table. Finally, applying Huffman coding to the *transform ID* further reduces memory usage by 4% to 8% depending on the scene.

6.3 Overall Compression Ratio

Table 4 compares our memory usage with a Sparse Voxel Octree (SVO), Sparse Voxel Directed Acyclic Graph (SVDAG) and a Symmetry-Aware Sparse Voxel Directed Acyclic Graph (**S**SVDAG). The Sparse Voxel Octree is encoded by storing the children of a node in consecutive memory. It requires that a node only stores a pointer to the first child; the remaining children can be accessed with a simple offset. As commonly used in the literature, we store these nodes using 64 bits: a 32-bit

Table 4. Memory usage of an SVO, SVDAG, **S**SVDAG, and our method. We present our method with symmetry S, axis permutation A, and translation T. "$S + A(B)$" uses the "base" pointer encoding (no look-up tables or Huffman coding), the other columns use our full encoding scheme. A dense voxel grid would require 35TB.

Scene	SVO	SVDAG	**S**SVDAG	S	A	S + A (B)	S + A	S + A + T
Bistro ($64k^3$) 21.4B voxels	13537MB (+1456%)	1309MB (+50%)	870MB (100%)	838MB (-4%)	890MB (+2%)	803MB (-8%)	764MB (-12%)	**721MB** **(-17%)**
Bunny ($64k^3$) 15.2B voxels	10137MB (+838%)	1736MB (+61%)	1080MB (100%)	1014MB (-6%)	1067MB (-1%)	900MB (-17%)	864MB (-20%)	**663MB** **(-39%)**
Crown ($64k^3$) 38.6B voxels	25647MB (+866%)	4146MB (+56%)	2654MB (100%)	2500MB (-6%)	2708MB (+2%)	2286MB (-14%)	2182MB (-18%)	**1851MB** **(-30%)**
Citadel ($64k^3$) 17.2B voxels	11430MB (+1126%)	1454MB (+56%)	932MB (100%)	840MB (-10%)	935MB (+0%)	787MB (-16%)	749MB (-20%)	**597MB** **(-36%)**
Lucy ($64k^3$) 6.3B voxels	4213MB (+526%)	1264MB (+88%)	673MB (100%)	645MB (-4%)	674MB (+0%)	597MB (-11%)	561MB (-17%)	**535MB** **(-21%)**
Dragon ($64k^3$) 6.3B voxels	4191MB (+516%)	1280MB (+88%)	681MB (100%)	655MB (-4%)	675MB (-1%)	605MB (-11%)	571MB (-16%)	**540MB** **(-21%)**

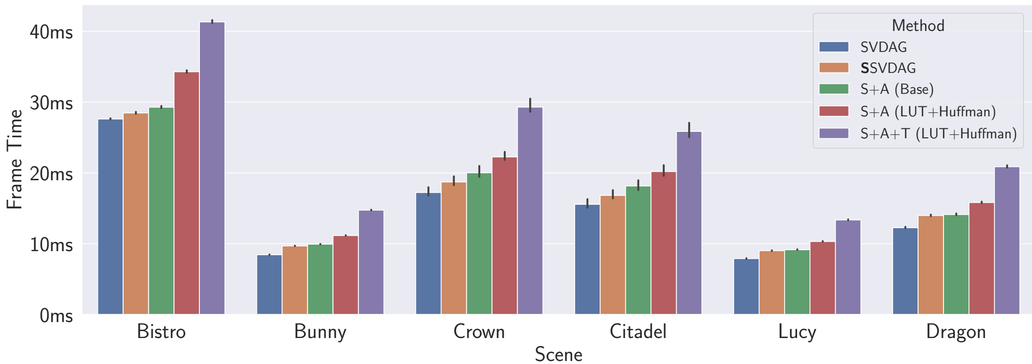


Fig. 10. Path-traced rendering performance comparison between a traditional SVDAG with 32-bit encoding, **S**SVDAG, and various configurations of our method.

pointer plus a child mask. For SVDAG [Kämpe et al. 2013] and **S**SVDAG [Villanueva et al. 2016], we use the encodings described in their respective papers.

Despite the efficient SVO encoding that reduces the number of bits per node, the SVO still requires significantly more memory than an SVDAG. In comparison, the additional transformations supported by our method, combined with our encoding scheme, provide significant memory savings over a traditional SVDAG [Kämpe et al. 2013]. Compared with the previous best method, **S**SVDAG, we see a typical improvement between 20% and 30% depending on the scene.

6.4 Render Performance

We measure rendering performance by capturing the frame time of a 1080p path-traced image with up to four random bounces. The scenes are lit by a (constant) environment light without next-event estimation. Figure 10 shows the results as measured on an NVIDIA RTX4070.

As expected, an SVDAG with fixed 32-bit pointer encoding outperforms variable pointer length methods in render time; typically by about 10%. With only child-reordering transformations (no translations), our method performs slightly worse than **S**SVDAG when we disable the pointer encoding features that require additional memory accesses (pointer look-up tables and Huffman). Enabling these encoding features increases the frame time by about 12%. Translations add an additional cost of about 30%. We theorize that most of this cost is caused by having to maintain an additional, but short, traversal stack (Section 5.3).

The increased rendering cost of **S**SVDAG and our method stem from the mechanisms to save memory and are linked to two main aspects. First, the GPU memory system was not designed to efficiently read variable-length pointers. Second, the larger traversal stack leads to either additional registry pressure or registry spilling. Given these observations, we suspect that the relative rendering cost of **S**SVDAG and our method will be lower on a CPU renderer.

7 Conclusion

Sparse Voxel Directed Acyclic Graphs (SVDAGs) have proven to be a very efficient method for storing sparse binary voxel data, while still enabling efficient random-access and ray intersections. The SVDAG is constructed from a Sparse Voxel Octree (SVO) by fusing similar subtrees. Previous

work shows how additional compression can be achieved by finding similarities via mirror transformations. However, the previous solution only considered a subset, and we suggested a simple extension (Section 3.4).

We then generalized the framework for efficiently finding subtrees under any transformation, which can be described as a (recursive) reorder of child pointers. We have shown how a limited subset of these transformations can be used to achieve practical compression. However, not all transformations can be described as permutations of a tree structure, and we explored translations as an example. Currently, finding these translations is still expensive due to the very large search space. However, we believe that this is an interesting area for future research. When combined with our novel pointer compression scheme, we achieve a typical improvement of 20% to 35% over the state-of-the-art [Villanueva et al. 2016].

Acknowledgments

This work has been partially supported by TKI HTSM PGT 2019 CREW.

References

Michal Aharon, Michael Elad, and Alfred Bruckstein. 2006. K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on signal processing* 54, 11 (2006), 4311–4322.

Rafael Ballester-Ripoll, Peter Lindstrom, and Renato Pajarola. 2019. TTHRESH: Tensor compression for multidimensional visual data. *IEEE transactions on visualization and computer graphics* 26, 9 (2019), 2891–2903.

David Benson and Joel Davis. 2002. Octree textures. *ACM Transactions on Graphics (TOG)* 21, 3 (2002), 785–790.

Boost. 2024. Boost C++ Libraries. http://www.boost.org/. Last accessed 2024-01-10.

Victor Careil, Markus Billeter, and Elmar Eisemann. 2020. Interactively modifying compressed sparse voxel representations. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 111–119.

Jiawen Chen, Dennis Bautembach, and Shahram Izadi. 2013. Scalable real-time volumetric surface reconstruction. *ACM Trans. Graph.* 32, 4 (2013), 113–1.

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*. 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.

David DeBry, Jonathan Gibbs, Devorah DeLeon Petty, and Nate Robins. 2002. Painting and rendering textures on unparameterized models. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 763–768.

Dan Dolonius, Erik Sintorn, Viktor Kämpe, and Ulf Assarsson. 2017. Compressing color data for voxelized surface geometry. *IEEE transactions on visualization and computer graphics* 25, 2 (2017), 1270–1282.

Volker Gaede and Oliver Günther. 1998. Multidimensional access methods. *ACM Computing Surveys (CSUR)* 30, 2 (1998), 170–231.

Pascal Gautron. 2022. Advances in Spatial Hashing: A Pragmatic Approach towards Robust, Real-time Light Transport Simulation. In *ACM SIGGRAPH 2022 Talks* (Vancouver, BC, Canada) *(SIGGRAPH '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 2 pages. https://doi.org/10.1145/3532836.3536239

Enrico Gobbetti, José Antonio Iglesias Guitián, and Fabio Marton. 2012. COVRA: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 1315–1324.

Stefan Guthe and Michael Goesele. 2016. Variable length coding for GPU-based direct volume rendering. In *Proceedings of the Conference on Vision, Modeling and Visualization*. 77–84.

Rama Karl Hoetzlein. 2016. GVDB: Raytracing sparse voxel database structures on the GPU. In *Proceedings of High Performance Graphics*. 109–117.

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

Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. 2013. High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–13.

Richard M Karp and Michael O Rabin. 1987. Efficient randomized pattern-matching algorithms. *IBM journal of research and development* 31, 2 (1987), 249–260.

Doyub Kim, Minjae Lee, and Ken Museth. 2024. NeuralVDB: High-resolution sparse volume representation using hierarchical neural networks. *ACM Transactions on Graphics* 43, 2 (2024), 1–21.

Yeojin Kim, Byungmoon Kim, and Young J Kim. 2018. Dynamic deep octree for high-resolution volumetric painting in virtual reality. In *Computer Graphics Forum*, Vol. 37. Wiley Online Library, 179–190.

Samuli Laine and Tero Karras. 2010. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics* 17, 8 (2010), 1048–1059.

Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*. 137–143.

Sylvain Lefebvre and Hugues Hoppe. 2006. Perfect spatial hashing. *ACM Transactions on Graphics (TOG)* 25, 3 (2006), 579–588.

Branislav Madoš and Norbert Ádám. 2021. Transforming hierarchical data structures–a psvdag–svdag conversion algorithm. *Acta Polytechnica Hungarica* 18, 8 (2021), 47–66.

Alistair Moffat. 2019. Huffman Coding. *ACM Comput. Surv.* 52, 4, Article 85 (Aug. 2019), 35 pages. https://doi.org/10.1145/3342555

Mathijs Molenaar and Elmar Eisemann. 2023. Editing Compressed High-resolution Voxel Scenes with Attributes. In *Computer Graphics Forum*, Vol. 42. Wiley Online Library, 235–243.

Mathijs Molenaar and Elmar Eisemann. 2024. Editing Compact Voxel Representations on the GPU. (2024).

Guy M Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. (1966).

Thomas Müller, Markus Gross, and Jan Novák. 2017. Practical path guiding for efficient light-transport simulation. In *Computer Graphics Forum*, Vol. 36. Wiley Online Library, 91–100.

Shigeru Muraki. 1993. Volume data and wavelet transforms. *IEEE Computer Graphics and applications* 13, 4 (1993), 50–56.

Ken Museth. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM transactions on graphics (TOG)* 32, 3 (2013), 1–22.

Ken Museth. 2021. NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks*. 1–2.

Eric Parker and Tushar Udeshi. 2003. Exploiting self-similarity in geometry for voxel based solid modeling. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*. 157–166.

Michael O Rabin. 1981. Fingerprinting by random polynomials. *Technical report* (1981).

Leonardo Scandolo, Pablo Bauszat, and Elmar Eisemann. 2016. Compressed Multiresolution Hierarchies for High-Quality Precomputed Shadows. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 331–340.

Erik Sintorn, Viktor Kämpe, Ola Olsson, and Ulf Assarsson. 2014. Compact precomputed voxelized shadows. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–8.

Herman Hansson Söderlund, Alex Evans, and Tomas Akenine-Möller. 2022. Ray Tracing of Signed Distance Function Grids. *Journal of Computer Graphics Techniques Vol* 11, 3 (2022).

Remi van der Laan, Leonardo Scandolo, and Elmar Eisemann. 2020. Lossy geometry compression for high resolution voxel scenes. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3, 1 (2020), 1–13.

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*. 7–14.

Brent Williams. 2015. *Moxel DAGs: Connecting material information to high resolution sparse voxel DAGs*. California Polytechnic State University.

Boon-Lock Yeo and Bede Liu. 1995. Volume rendering of DCT-based compressed 3D scalar data. *IEEE Transactions on Visualization and Computer Graphics* 1, 1 (1995), 29–43.