Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes

Alberto Jaspe Villanueva
CRS4

Fabio Marton
CRS4

Enrico Gobbetti
CRS4

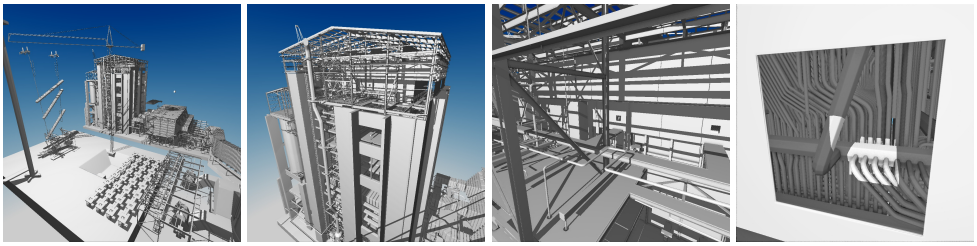


Figure 1. The *PowerPlant* scene voxelized to a $256K^3$ resolution and stored as a Symmetry-aware Sparse Voxel DAG (SSVDAG). The total non-empty voxel count is nearly 100 billion, stored in less than 575 MB at 0.048 bits/voxel. A sparse voxel octree would require 31.1 GB without counting pointers, over 55 times more, while a Sparse Voxel DAG would require 1.0 GB, nearly double. Primary rays as well as hard shadow are raytraced directly in the SSVDAG structure, and shading normals are estimated in screen space.

Abstract

Voxelized representations of complex 3D scenes are widely used to accelerate visibility queries in many GPU rendering techniques. Since GPU memory is limited, it is important that these data structures can be kept within a strict memory budget. Recently, directed acyclic graphs (DAGs) have been successfully introduced to compress sparse voxel octrees (SVOs), but they are limited to sharing identical regions of space. In this paper, we show that a more efficient lossless compression of geometry can be achieved while keeping the same visibility-query performance. This is accomplished by merging subtrees that are identical through a similarity transform and by exploiting the skewed distribution of references to shared nodes to store child pointers using a variabile bit-rate encoding. We also describe how, by selecting plane reflections along the main grid directions as symmetry transforms, we can construct highly compressed GPU-friendly structures using a fully out-of-core method. Our results demonstrate that state-of-the-art compression and real-time tracing performance can be achieved on high-resolution voxelized representations of real-world scenes of very different characteristics,

 ISSN 2331-7418

including large CAD models, 3D scans, and typical gaming models, leading, for instance, to real-time GPU in-core visualization with shading and shadows of the full Boeing 777 at sub-millimeter precision. This article is based on an earlier work: *SSVDAGs: Symmetry-aware Sparse Voxel DAGs, in Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games ©ACM, 2016. http://dx.doi.org/10.1145/2856400.2856420.* We include here a more thorough exposition, a description of alternative construction and tracing methods, as well as additional results. In order to facilitate understanding, evaluation and extensions, the full source code of the method is provided in the supplementary material.

1. Introduction

With the increase in performance and programmability of graphical processing units (GPUs), GPU raycasting is emerging as an efficient solution for many real-time rendering problems. In order to handle large detailed scenes, devising compact and efficient scene representation for accelerating ray-geometry intersection queries becomes paramount, and many solutions have been proposed (see Sec. 2). Among these, sparse voxel octrees (SVO) [Laine and Karras 2011] have provided impressive results, since they can be created from a variety of scene representation; they efficiently carve out empty space, with benefits on ray tracing performance and memory needs; and they implicitly provide a level-of-detail (LOD) mechanism. Given their still relatively high memory cost, and the associated high memory bandwidth required, these voxelized approaches have, however, been limited to moderate scene sizes and resolutions, or to effects that do not require precise geometric details (e.g., soft shadows). While many extremely compact representations for high-resolution volumetric models have been proposed, especially in the area of volume rendering [Balsa Rodríguez et al. 2014], the vast increase in compression rates of these solutions is balanced by increased decompression and traversal costs, which makes them hardly usable in general settings. This has triggered a search for simpler scene representations that can provide compact representations within reasonable memory footprints, while not requiring decompression overhead. Kämpe et al. [2013] have recently shown that, for typical video-gaming scenes, a binary voxel grid can be represented orders of magnitude more efficiently than using a SVO by simply merging together identical subtrees, generalizing the sparse voxel tree to a directed acyclic graph (SVDAG). Such a representation is compact, as nodes are allowed to share pointers to identical subtrees, and remains as fast as SVOs and simple octrees, since the tracing routine is essentially unchanged.

Our approach In this work, we show that efficient lossless compression of geometry can be combined with good tracing performance by merging subtrees that are identical up to a similarity transform, using different granularity at inner and leaf nodes, and compacting node pointers according to their occurrence frequency. The resulting structure, dubbed *Symmetry-aware Sparse Voxel DAG* (SSVDAG) can be efficiently

constructed by a bottom-up external-memory algorithm that reduces an SVO to a minimal SSVDAG by alternating different phases at each level. First, all nodes that represent similar subtrees are clustered and replaced by a single representative. Then, pointers to those nodes in the immediately higher level are replaced by tagged pointers to the single representative, where the tag encodes the transformation that needs to be applied to recover the original subtree from the representative. Finally, representatives are sorted by their reference count, which allows for an efficient variable-bit-rate encoding of pointers. We show that, by selecting planar reflections along the main grid directions as symmetry transform, good building and tracing performance can be achieved.

Contribution Our main contributions are:

- A compact representation of a Symmetry-aware Sparse Voxel DAG that can losslessly represent a voxelized geometry of many real-world scenes within a small footprint and can be efficiently traced;
- An out-of-core algorithm to construct such representation from a SVO or a SVDAG; we describe, in particular, a simple multi-pass method based on repeated merging operations;
- A clean modification of standard GPU raycasting algorithm to traverse and render this representation with small overhead. We describe, in particular the details of a GPU tracing method based on a multi-resolution Digital Differential Analyzer (DDA), implemented with a full stack. This sort of approach has been proven effective in previous work on SVOs and SVDAGs [Laine and Karras 2011; Kämpe et al. 2013], and is extended here to handle graphs with reflective transformations.

This work is an invited extended version of our I3D 2016 paper [Jaspe Villanueva et al. 2016]. We here provide a more thorough exposition, but also significant new material, including the presentation of an alternative simpler construction method, the description of alternative tracing methods, and additional qualitative and quantitative results. Finally, we have attempted to further clarify the steps in our algorithms to facilitate their implementation and to make the transfer between abstract concepts and actual code as straightforward as possible. To this end, a reference implementation of the method, supporting voxelization and conversion to SSVDAG format, as well as rendering using OpenGL, is provided as supplemental material.

Advantages and limitations Our reduction technique is based on the assumption that the original scene representations is geometrically redundant, in the sense that it contains a large number of subtrees which are similar with respect to a reflective transformation. Our results, see Sec. 6, demonstrate that this assumption is valid for

real-world scenes of very different characteristics, ranging from large CAD models, to 3D scans, to typical gaming models. This makes it possible to represent very large scenes at high resolution on GPUs, and to support precise geometric rendering and high-frequency phenomena, such as sharp shadows, with a tracing overhead of less than 15%. Similarly to other works on DAG compression [Kämpe et al. 2013; Sintorn et al. 2014; Kämpe et al. 2015], we focus in this paper only on geometry, and not on non-geometric properties of voxels (e.g., material or reflectance properties), which should be handled by other means. A recent example on how to associate attributes to the original SVDAGs has for instance been recently presented by Dado et al. [2016].

2. Related Work

Describing geometry for particular applications and devising compressed representation of volumetric models are broad research fields. Providing a full overview of these areas is beyond the scope of this paper. We concentrate here on methods that employ binary voxel grids to represent geometry to accelerate queries in GPU algorithms We refer the reader to a recent survey [Balsa Rodríguez et al. 2014] for a more general overview in GPU-friendly compressed representations for volumetric data.

Starting from more general bricked representations proved successful for semi-transparent GPU raycasting [Gobbetti et al. 2008; Crassin et al. 2009], Laine and Karras [2011] have introduced Efficient Sparse Voxel Octrees (ESVOs) for raytracing primary visibility. In their work, in addition to employing the octree hierarchical structure to carve out empty space, they prune entire subtrees if they determine that they are well represented by a planar proxy called a contour. Storing the proxy instead of subtrees achieves considerable compression only in scenes with many planar faces, and introduces stitching problems as in other discontinuous piecewise-planar approximations [Agus et al. 2010]. Crassin et al. [2011] have shown the interest of such approaches for secondary rays, computing ambient occlusion and indirect lighting by cone tracing in a sparse voxel octree. Their bricked structure, however, requires large amounts of memory, also due to data duplication at brick boundaries.

A number of works have thus concentrated on trying to reduce memory consumption of such voxelized structures while maintaining a high tracing performance. Crassin et al. [2009] mentioned the possibility of instancing, but rely on ad-hoc authoring for fractal scenes, rather than algorithmic conversions. Compression methods based on merging common subtrees have been originally employed in 2D for the lossless compression of binary cartographic images [Webber and Dillencourt 1989], and extended to 3D by Parker and Udeshi [2003] to compress voxel data. These algorithms, however, are costly and require fully in-core representations of voxel grids. Moreover, since voxel content is not separated from voxel attributes, only moderate compression is achieved. Recently, Dado et al. [2016] presented a compressed structure able to

encode/decode voxels attributes, such as color or normals in another auxiliary structure. This work is orthogonal to ours. Hoetzlein [2016] created GVDB, a hierarchy of grids of volumetric models with dynamic topology, with the scope of fast decoding in GPU. The traversal algorithm is similar to our DDA-based one.

High Resolution Sparse Voxel DAGs (SVDAG) [Kämpe et al. 2013] generalize the trees used in Sparse Voxel Octrees (SVOs) to DAGs, allowing the sharing of common octrees. They can be constructed using an efficient bottom-up algorithm that reduces an SVO to a minimal SVDAG, which achieves significantly reduced node count even in seemingly irregular scenes. The effectiveness of the method is demonstrated by ray-tracing high-quality secondary-ray effects using GPU raycasting from GPU-resident SVDAGs. This approach has later been extended to shadowing by voxelizing shadow volumes instead of object geometry [Sintorn et al. 2014; Kämpe et al. 2015], as well as for time-varying data [Kämpe et al. 2016]. We improve over SVDAGs by merging subtrees that are identical up to a similarity transform, and present an efficient encoding and building algorithm, with an implementation using reflective transformations. The idea of using self-similarity for compression has also found application in point cloud compression [Hubo et al. 2008], where, however, the focus was on generating approximate representations instead of lossless ones.

In addition to reducing the number of nodes, compression can be achieved by reducing node size. As pointers are very costly in hierarchical structures, a number of proposals have thus focused on reducing their overhead. While pointerless structures based on exploiting predefined node orderings have been proposed for offline storage [Schnabel and Klein 2006], they do not support efficient run-time traversal. The optimizations used for trees, such as grouping children in pages and using relative indexing within pages [Laine and Karras 2011; Lefebvre and Hoppe 2007] are not applicable to our DAGs, since children are scattered throughout the structure due to sharing. By taking advantage of the fact that the reference count distribution of shared nodes is highly skewed, we employ a simple variable bit-rate encoding of pointers. A similar approach has been used by Dado et al. [2016] for compression of pointers in their auxiliary structures for storing voxel attributes. That system has been independently developed in parallel to ours.

3. Overview

A 3D binary volumetric scene is a discretized space subdivided in N^3 cells called voxels, which can be empty or full. Since this structure grows cubically for every subdivision, is is hard to achieve high resolutions. SVOs compactify these representation using a hierarchical octree structures of nodes arranged in a number of levels (L), with $N = 2^L$, and most commonly represented using a children bitmask per node as well as up to eight pointers to nodes in the next level. When one of those children represents an

empty area, no more nodes are stored under it, introducing sparsity and thus efficiently encoding whole empty areas of the scenes. The structure can be efficiently traversed on the GPU using stackless or short-stack algorithms [Laine and Karras 2011; Beyer et al. 2015], which exploit sparsity for efficient empty-space skipping.

SVOs and grids can be directly created from a surface representation of the scene through a *voxelization* process, for which many optimized solutions have been presented (see, e.g., Crassin and Green [2012]). In this paper, we use a straightforward CPU algorithm that builds SVOs using a streaming pass over a triangle soup, inserting triangles in an adaptive octree maintained out-of-core using memory-mapped arrays. Using other more optimized solutions would be straightforward.

SVDAGs optimize SVOs by transforming the tree to a DAG, using an efficient bottom-up process that iteratively merges identical nodes one level at a time and then updates the pointers of the level above. The resulting structure is more compact than SVOs, and can be traversed using the exact same ray-casting algorithm, since node sharing is transparent to the traversal code.

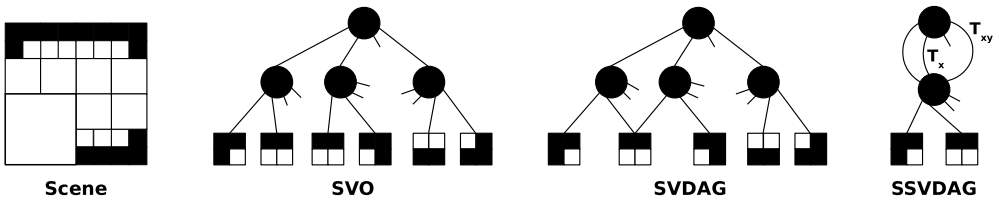


Figure 2. Example 2D scene transformed into different structures, with children ordered left to right, top to bottom. The Sparse Voxel Octree (SVO) contains 10 nodes. The Sparse Voxel Directed Acyclic Graph (SVDAG) finds one match and then shares a node, meaning 9 nodes. The presented Symmetry-aware Sparse Voxel Directed Acyclic Graph (SSVDAG) finds reflective matches in two last levels, and reduces the structure to 4 nodes.

The aim of this work is to obtain a more compact representation of the volume, while keeping the efficiency in traversal and rendering. We do this by merging self-similar subtrees (starting from an SVDAG or an SVO), and by reducing node size through an adaptive encoding of children references.

Among the many possible similarity transformations, we have selected to look for *reflective symmetries*, i.e. mirror transformations along the main grid planes. We thus consider two subtrees similar (and therefore merge them) if their content is identical when transformed by any combination of reflections along the principal planes passing through the node center. Such a transformation $T_{x,y,z}$ has the advantage that the 8 possible reflections can be encoded using only 3 bits (reflection X,Y,Z), that the transformation ordering is not important, as transformations along one axis are independent from the others, and that efficient access to reflected subtrees, which requires application of the direct transformation $T_{x,y,z}$ or of its inverse $T^{-1}_{x,y,z} = T_{x,y,z}$, can be achieved by simple coordinate (or index) reflection. This leads to

efficient construction (see Sec. 4.1) and traversal (see Sec. 5). In addition, since the transformation has a geometric meaning, the expectation, verified in practice, is to frequently find mirrored content in real-world scenes (see a 2D example in Figure 2). The output of the merging process is a DAG in which non-empty nodes are referenced by tagged pointers that encode the transformation $T_{x,y,z}$ that needs to be applied together with the child index. Further compression is achieved by taking advantage of the observation that not all subtrees are uniformly shared, i.e., some subtrees are significantly referenced more than others. We thus use a variable bit-rate encoding, in which the most commonly shared subtrees are referenced with small indexes, while less common subtrees are referenced with more bits. This is achieved through a per-level node reordering process, followed by a replacement of child pointers by indices. The encoding process, as well as the resulting final encoding is described in Sec. 4.3.

4. Construction and encoding

An SSVDAG is constructed bottom-up starting from a voxelized representation (SVDAG or SVO). We first explain how a minimal SSVDAG is constructed by merging similar subtrees, and then explain how the resulting representation is compactly encoded in a GPU-friendly structure. While the original work [Jaspe Villanueva et al. 2016] described a fully out-of-core implementation based on external-memory arrays, we propose a simpler approach in this paper that proceeds by repeated in-core reductions of subtrees.

4.1. Bottom-up construction process

Constructing the SSVDAG requires efficiently finding reflectively-similar subtrees. Since explicitly checking similarity in subtrees would be prohibitively costly for large datasets, we use a bottom-up process that iteratively merges similar nodes one level at a time. This requires some important modifications to the original SVDAG construction method. In our technique, we use arrays encoding the existing nodes level-by-level, with one array per level. For construction, each array element contains an uncompressed node description containing a bitmask for leaf nodes, while inner nodes contain 8 (possibly null) child pointers and a bitmask to take into account invariance with respect to transformation during inner nodes clustering (see below). We start the construction process from the finest level $L-1$, and proceed up to the root at level 0.

Our construction code is capable of performing a transformation into a DAG with or without symmetries, and works, for compatibility with previous encoding methods, using a leaf size of 2^3. Grouping into larger leaves is performed in post-processing during our encoding phase (see Sec. 4.3). At each level, we first group the nodes into clusters of self-similar nodes, then select one single representative per cluster and

associate the transformation for each of the others that maps them to the representative. The surviving nodes are reordered for compact encoding (see Sec. 4.3) and stored in the final format. Child pointers of nodes at the previous level are then updated to point to the representatives, and the process is repeated for all levels up to the root.

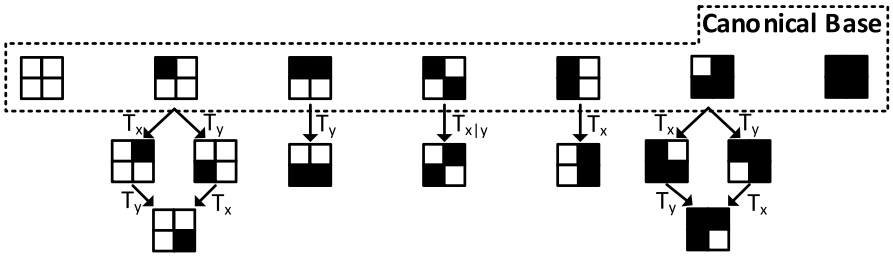


Figure 3. 2D example of canonical transformations of a small voxel grid into a set of base representatives. The transformation maps clusters of self-similar grids to a unique representative.

The matching process at the core of clustering is based on the concept of reordering the nodes at a given level so that matching candidates are stored nearby. Clustering and representative selection is then performed during a streaming pass. Leaf nodes and inner nodes, however, must use different methods to compute ordering and perform matching.

Leaf node clustering In order to efficiently match leaf nodes in the tree, we must discover which representation of small voxel grids remain the same when one of the possible transformations is applied. Considering that each grid G can be represented by a binary number $B(G)$ by concatenating all the voxel occupancy bits in linear order, we define a mapping of each possible grid G to another grid $G^\star = T^\star_{x,y,z}(G)$, such that the *canonical transformation* $T^\star_{x,y,z} = \arg\max_{T_{x,y,z}} B(T_{x,y,z}(G))$. $G^\star$ is the *canonical representation* of G, and represents, among all possible reflections of G, the one with the largest integer value. Geometrically, it is the one that attracts most of the empty space to the origin (see Figure 3). This transformation is precomputed in a table of 256 entries that maps all the possible combinations of 2^3 voxels to the bitcode representing the canonical transformation as well as to the unique canonical representation (one of the 46 possible ones). Given this transformation, two nodes are self-similar if their canonical representation is the same. Clustering can thus be performed in a single streaming pass after sorting leaves using the canonical representation as a key. Nearby nodes sharing the same canonical representation are merged into a single representative, pointers at the upper level are then updated to point to the representative, and pointer tags are computed so as to obtain the original leaf from the representative.

Inner node clustering While for leaf nodes we can detect symmetries by directly looking at their bit representation, two inner nodes n_1 and n_2 must be merged if they

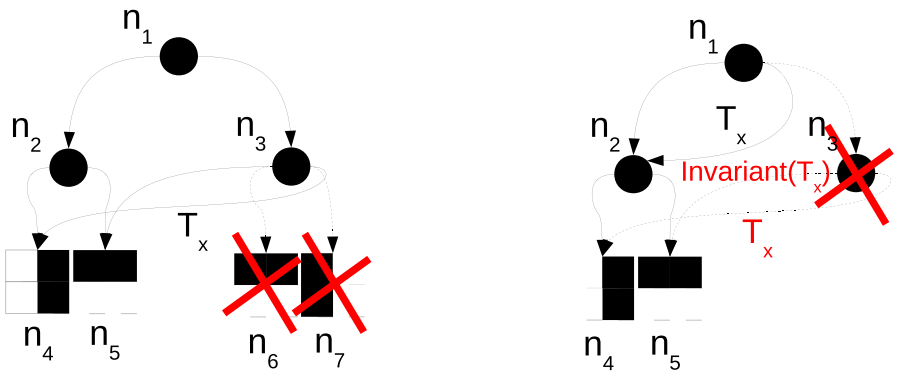


Figure 4. On the left, during leaf clustering, references to leaf node n_6 are replaced by references to n_5, which is identical, while references to n_7 are replaced with references to n_4 transformed by transformation T_x. On the right, inner node n_3 is replaced with n_2 through transformation T_x since its left child n_5 is invariant to transformation T_x and is identical to the right child of n_2, while its right child matches the left child of n_2 through the same transformation T_x.

represent the exact same *subtrees* when a transformation T is applied. The first trivial condition to be checked is that child pointers must be the same. We thus sort the inner nodes using the lexicographically sorted set of pointers to children as the key. After sorting, all self-similar nodes are positioned nearby, as they are among those that share the same set of pointers. During a streaming pass, we perform merging by creating one representative per group of self-similar nodes. The self-similarity condition must be verified without performing a full subtree comparison. Given the properties of our reflective transformations, we have thus to verify that, when the two nodes n_1 and n_2 are matched for similarity under a candidate transformation T, every tagged pointer (tag, p) of n_1 is mapped to $(T(tag), p)$ in n_2 if p is not invariant to the transformation T, or it is mapped to $(T(tag) \bigvee \neg T(tag), p)$ if it is invariant (see Figure 4). This means, for instance, that, when looking for a match under a left-right transformation T_x, the left and right pointers must be swapped in n_2 with respect to n_1, and the pointed subtrees must be equal under a left-right mirroring, The latter condition is verified if the pointed subtree has a left-right symmetry, or if, for each matched pair of tagged child pointers, the left-right transformation bit is inverted while the other bits are the same. This process does the clustering for one particular transformation T, and is repeated for each of the 8 possible reflection combinations, stopping at the first transformation that generates a match with one of the currently selected representatives or creating a new representative if all tests are unsuccessful. In order to efficiently implement invariance checks, we thus associate three invariant bits (one for each mirroring direction) at each of the leaves when computing their canonical representations, and pull them up during construction at inner nodes by suitably combining the invariant bits of pointed nodes at each merging step. For instance, an inner node is considered invariant with

respect to a left-right transformation if all its children are invariant with respect to that transformation, or the left children are the mirror of the right ones.

4.2. Out-of-core implementation

The construction process described above constructs an SSVDAG starting from a voxelized representation (SVO or SVDAG), and must be made scalable to massive models.

Our original work [Jaspe Villanueva et al. 2016] used a direct implementation of the described method, which performed the reduction in a single pass, using external memory structures to store, by level, all the required data. These structures were per-level memory-mapped arrays storing the voxel tree. Scalability was thus achieved by relying on operating system features to handle virtual memory. This required large (out-of-core) temporary storage space to store the fully constructed tree.

In this work. we instead employ a simpler solution based on building and merging bottom-up portions of the dataset fully fitting in core memory.

It should be noted that in this paper the goal of SSVDAG reduction is to produce, even from very massive inputs, an end results that needs to fit within GPU memory constraints. A recursive merging solution, which reduces voxelized representations bottom-up until they fit into memory thanks to partial reduction is therefore applicable. Even if such a construction approach assumes that the final reduction can be performed fully in-core, this is a safe assumption in our context, since the final reduced SSVDAG produces a data structure that should be stored in GPU memory, which is typically much smaller than available RAM.

When starting from a model whose representation exceeds the available memory for construction, we therefore construct an SSVDAG of depth L using the following process:

1. We estimate the level $L_0 < L$ of the octree structure at which all subtrees are deemed small enough to fit into memory once transformed into SSVDAGs.

2. For each of the voxels at level L_0, we perform a separate and independent reduction process confined to their spatial regions by applying in sequence the following steps:

   (a) We create, in-core, a voxelized representation of depth $L - L_0$ of the portion of the model contained within the voxel. If the representation is already available, the relevant voxel data is just loaded from disk, otherwise it is computed on-the-fly through a voxelization process. In the latter case, for a mesh, this can be done by streaming over the input model's triangles, without the need to load the entire model in memory.

   (b) We reduce, in-core, the voxelized representation to an SSVDAG using the process described in Sec. 4.1, and we store the resulting SSVDAG on disk.

3. In order to compute the final DAG of depth L, we recursively merge the reduced representations computed in the previous phase bottom-up. For each reduction step, this is achieved by:

   (a) We load all the SSVDAGS associated to the voxels of the relevant subtrees in memory and construct a single DAG by creating octree nodes above them until the common root is reached.

   (b) We apply the reduction process described in Sec. 4.1 to the union of SSVDAGS instead of an octree, and we store the resulting SSVDAG on disk.

4. We repeat the process bottom up, until we have reduced the root of the original octree to a single SSVDAG; at this point, the stored SSVDAG is the final model.

Separately processing subtrees (or sub-DAGs) during the reduction phases performed at a given octree level will reduce voxel counts by finding symmetries only in the spatial region represented by each subtree, which are a subset of the total ones. It is important to note that this is not a limitation, since the overall global reduction will be obtained when the separately reduced subtrees are merged in subsequent bottom-up reduction phases. This is because each subtree reduction always restarts from the leaf level L of the merged subtrees (or sub-DAGS) and recurses up to their root. We exploit this fact to parallelize the construction process, in order to perform several reductions in parallel. On a machine with *num_procs* processors, we thus give as maximum memory budget for subtree construction the maximum available in-core memory divided by *num_procs*, and then perform *num_procs* reductions in parallel.

4.3. Compact encoding

The outlined construction process produces a DAG, where inner nodes point to children through tagged pointers that reference a child and encode the transformation that has to be applied to recover the original subtree. We encode such a structure in a GPU friendly format aimed at reducing the pointer overhead, while supporting fast tracing without decompression. We achieve this goal through leaf grouping, frequency-based pointers compaction, and memory-aligned encoding.

Leaf grouping While a leaf size of 2^3 allows for an elegant construction method using table-based clustering, such a level of granularity leads to a high structure overhead. Rays have to traverse deep pointer structures to reach small 8-voxel grids, and the advantage of clustering is offset by the need to encode pointers to these small nodes. For final encoding, we have thus decided to coarsen the construction graph by one level, encoding as simple grids all the 4^3 grids, and to store them in a single array of bricks, each occupying 64 bits. Note that this decision does not require performing new matches on 4^3 leaves, since we just coarsen the graph obtained with the bottom-up

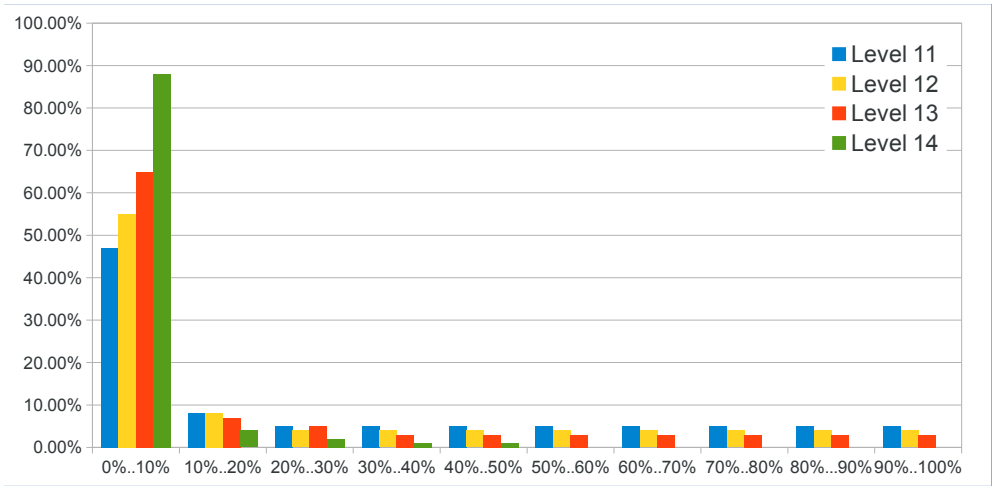


Figure 5. Histograms of the references to nodes in the Powerplant dataset voxelized at $64K^3$ resolution, with nodes sorted by reference counts. As we can see, the distribution is highly skewed, and the most popular 10% of the nodes account for most of the references.

process described in Sec. 4.1, which uses a table-based matching on 2^3 leaves at level $L-1$ to drive the construction of 4^3 inner nodes at level $L-2$.

Frequency-based pointer compaction We have verified that in our SSVDAG the distributions of references to nodes is highly skewed. This means that there typically is a small groups of node referenced by a lot of parents nodes, while many others are referenced much less. Figure 5, for instance, shows the histogram of the distribution of reference counts in the Powerplant dataset of Figure 1, where the most common 10% of the nodes is referenced by nearly 90% of pointers at level 14 (nearly 50% at level 11). We have thus adopted an approach in which frequently used pointers are represented with less bits than more frequent ones. In order to do that, for encoding, we reorder nodes at each level using the number of references to it as a key, so that most referenced nodes appear first in a level's array. We then replace pointers with offsets from the beginning of each level array, and chose for each offset the smallest number of bits available in our encoded format (see below).

Memory aligned encoding While leaf nodes are all of the same size (64 bits), the resulting inner node encoding produces variable-sized records (which is true also for other DAG formats with variable child count, e.g., SVDAG [Kämpe et al. 2013]). We have decided, in order to simplify decoding, to use half-words (16 bits) as the basis for our encoding. Our final encoding includes an indexing structure, an array of inner nodes, and an array of leaf nodes. The indexing structure contains the maximum level L and three 32-bits offsets in the inner level array that indicate the start of each level. The layout of inner-level nodes is depicted in Figure 6. For each node, we store in a 16-bit header a 2-bit code for each of the 8 potential children. Tag 00 is reserved

for null pointers, which are not stored, while the other tags indicate the format in which child pointers are stored after the header. Children of type 01 use 16 bits, with the leftmost 3 bits encoding the transformation, while the remaining 13 bits encode the offset in number of level-words from the beginning of the next level, where a level-word is 2 bytes for an inner level and 8 bytes for the leaf level. Thus, reflections and references to nodes stored in the first 2^{14} bytes of an inner level's array or in the first 2^{16} bytes of the leaf level can be encoded with just two bytes. Less frequent children pointers, associated to header tags 10 and 11, are both encoded using 32 bits, with the leftmost 3 bits encoding the transformation, and the rightmost ones the lowest $29bits$ of the offset. The highest bit of the offset is set to the rightmost bit of the header tag. We can thus address more than 2 GB into an inner level, and 8 GB into leaves.

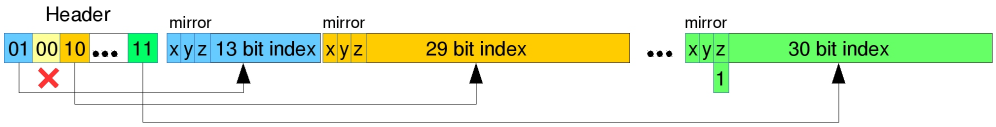


Figure 6. Layout of inner nodes in our compact representation.

5. Ray-tracing a SSVDAG

The SSVDAG structure can be efficiently traversed using a GPU-based raytracer by slightly adapting other octree-based approaches to apply the transformation upon entering a subtree. In order to test several approaches within the same code base, we have implemented a basic GPU-based raytracer to traverse both SVO, SVDAG, and our SSVDAG. While the structure alone, similar to SVDAG [Kämpe et al. 2013], is mostly useful for visibility queries and/or secondary rays effects, in order to fully test the structure in the simplest possible setting, we use the raytracer both from primary rays (requiring closest intersections) and for hard shadows for the view samples of a deferred rendering target. We focus on these effects, rather than soft shadows and ambient occlusion, since they are the ones where voxelization artifacts are most evident. In the following, we first explain how standard octree traversal code can be minimally transformed to support reflections, and then discuss integration in our sample raytracer to fully render geometric scenes.

5.1. Traversal

Many octree raycasting approaches can be adapted to trace SSVDAGs, which differ from regular SVOs or SVDAGs only because they need to apply reflection transformations every time a pointer is followed. As for all trees containing geometric transformations, this can be achieved by one of the two complementary approaches of applying the inverse transformation to the ray or the direct geometric transformation to

the voxel geometry.

5.1.1. Transforming the ray

The ray-transformation approach is extremely simple to integrate in stackless traversal approaches [Crassin et al. 2009], which cast rays against a regular octree using kd-restart algorithm [Foley and Sugerman 2005], and traverse encountered leaves as uniform 3D grids. In this case, once the tagged pointer to the child is traversed, the associated transformation is extracted, and ray reflection is obtained by conditionally performing, for each axis in which the transformation is applied, a mirroring with respect to the node's center c of the ray's current origin together with a sign change of the ray direction. This conditional code can be concisely implemented as follows:

```
org = c + s * (org-c)
dir *= s;
```

where $s \in \mathcal{R}^3$ is equal to -1 in the coordinates in which a mirroring is applied, and $+1$ otherwise. Since the node's center is already maintained in kd-restart algorithms during tree descent in order to locate leaves in the octree, the overhead of mirroring is thus mininal. Traversal stops when a non-empty voxel is found or the ray span is terminated.

In more optimized implementations, such as *pushdown* or *short-stack*, which avoid restarting from the root of the tree [Horn et al. 2007], in addition to applying this transformation one would store the modified local ray together with the restart node in the restart cache or stack. This could put more pressure on registers or local memory and reduce parallelization efficiency on GPUs. Moreover, handling changes of ray directions during traversal due to the reflections would increase book-keeping costs in methods not simply based on repeated octree point location, such as kd-restart.

5.1.2. Transforming the geometry

Directly transforming the geometry allows the raytracer to reduce book-keeping costs during traversal, and therefore leads to a more optimized implementation. In this work, we thus discuss how one can traverse our voxel structure using a depth-first visit of the SSVDAG based on a multi-resolution Digital Differential Analyzer (DDA), implemented with a full stack. This kind of DDA-based traversal approach has been proven effective in previous work on SVOs and SVDAGs [Laine and Karras 2011; Kämpe et al. 2013]. All the rendering results in this paper are obtained with this implementation.

Listing 1 succintly describes our traversal algorithm, and shows the modifications to the standard octree DDA traversal needed to navigate between nodes with symmetries, using only three extra bits to encode reflections (variable `mirror_mask` in the code).

This 3-bit transformation status indicates which reflections must be applied to the indices used to access child pointers in inner nodes or voxel contents in leaf nodes.

```
vec3 trace_ray(Ray r, float proj_factor) {
  t = ray.t_min; mirror_mask = 0; level = 0; cell_size = 0.5;

  (voxidx, ...) = DDA_init();

  node_idx = 0; leaf_data = 0;
  inner_hdr = fetch_inner_hdr(node_idx);
  mirrored_voxidx = mirror(mirror_mask, voxidx, level);

  do {
    bool is_full_voxel = (level<MAX_LEVEL) ?
    inner_voxel_bit(mirrored_voxidx, inner_hdr) :
    leaf_voxel_bit(mirrored_voxidx, leaf_data);

    if (!is_full_voxel) {
      // Empty - try to move forward at current level
      (voxidx, ...) = DDA_next();

      if (!is_in_bounds(voxidx, level)) {
        if (stack_is_empty()) {
          return NO_INTERSECTION;
        } else {
          // Move up and forward at upper level
          old_level = level:
          (node_idx, inner_hdr, mirror_mask, level) = stack_pop();
          cell_size *= (1 << (old_level-level));
          (voxidx, ...) = DDA_next();
        } // if stack empty
      } // if !in bounds
    } else {
      // Full - return intersection or refine
      if (level==MAX_LEVEL || resolution_ok(t, cell_size, proj_factor)) {
        return t; // INTERSECTION FOUND
      } else {
        // Go down
        if (is_in_bounds(DDA_next())) {
          stack_push(node_idx, inner_hdr, mirror_mask, level);
        }
        (m,node_idx)=fetch_tagged_ptr(inner_hdr,node_idx,mirrored_voxidx);
        mirror_mask ^= m;

        // Update DDA and fetch next node
        ++level; cell_size *= 0.5;
        (voxidx, ...) = DDA_down();
        if (level < MAX_LEVEL-1) {
          inner_hdr = fetch_inner_hdr(node_idx);
        } else {
          // Leaves are 4x4 - must refine DDA
          ++level; cell_size *= 0.5;
          (voxidx, ...) = DDA_down();
          leaf_data = fetch_leaf_data(node_idx);
        }
      }
    }

    mirrored_voxidx = mirror(mirror_mask, voxidx, level);
  } while (t < ray.t_max);

  return NO_INTERSECTION;
}
```

Listing 1. Optimized Octree DDA traversal for SSVDAGs.

The transformation status is initialized to 0 (line 2), is updated each time we descend in a child (line 40), and is pushed to the stack together with the current node to be able to restore it upon moving up in the hierarchy (line 37).

When descending, as in SVDAG [Kämpe et al. 2013], we must access the i-th children pointer by computing an offset within the header equal to the size of all pointers in the interval $[0, i-1]$, with the only difference that in SVDAG the only two size possibilities are 0 and 4, while in our case tagged pointers can be stored using 0, 2, and 4 bytes. This computation is performed in our shader using a manually unrolled loop within routine `fetch_tagged_ptr` in line 39. Once the tagged pointer to the child is found, the associated transformation code is given by the 3 highest bits, and should be applied to all the nodes in the subtree, which is achieved by xor-ing it with the current `mirror_mask`.

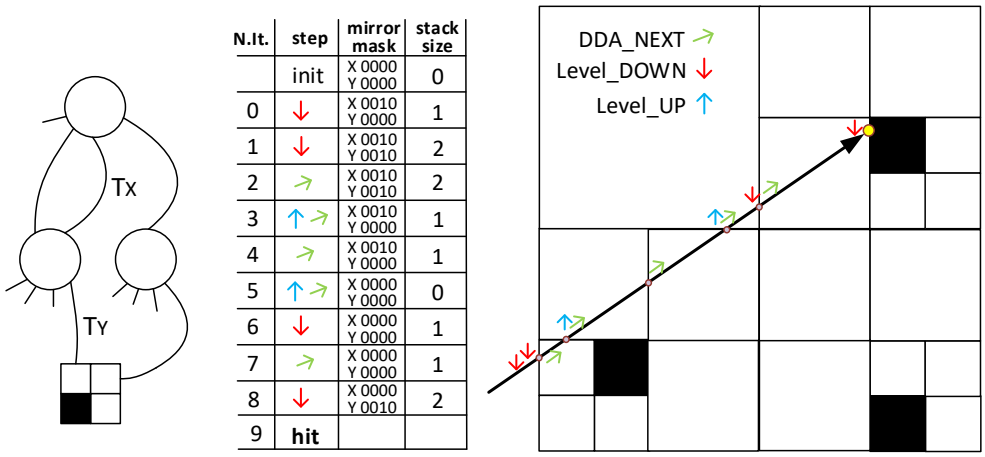
In order to support SSVDAGs, index reflection is applied when accessing the voxel occupancy bits at inner and leaf nodes, as well as child pointers associated to a non-empty inner node voxel (line 8) . This is done by transforming a local 3D index v to a node's voxel index that takes into account the current mirroring transformation (function `mirror` in the pseudo-code). This can be obtained by computing the mirrored coordinates as

$$((1,1,1) - 2 \cdot M) * v + M \cdot (S-1)$$

where M is the 3-bit mirroring vector, v the child coordinates, and S the cubical size of the voxel, that is 2 for inner nodes, and 4 for the leaves. The difference in size between inner and leaf nodes is also taken into account during the down phase, executing two `DDA_down()` steps instead of one at the last level (lines 47-51).

It should be noted that, in order to minimize memory pressure, similarly to previous work [Laine and Karras 2011], we don't push a node to the stack if the next DDA step would cause the ray to exit from it. This makes it possible to skip useless steps when moving back up in the hierarchy (lines 23-27), but forces us to also put the level in the stack. Moreover, in order to avoid refetching a parent node's header when going up from the child, the header contents are also pushed. With this organization, data is fetched from global memory only when going down, either to fetch the pointer to follow (line 39), or the data associated to inner node (line 46) or leaf node (line 51). Figure 7 shows an example execution of the traversal algorithm, illustrating a full traversal with conditional stack push.

During traversal, moreover, we maintain the voxel cell size, which is updated at the initialization (line 2), when moving up (line 26), and when moving down (line 43 as well as line 49 for leaf nodes). Maintaining this size makes it possible to stop the traversal when the projected size of a voxel goes below a threshold (line 32), which makes it possible to effectively implement levels of detail.



N.It.	step	mirror mask	stack size
	init	X 0000 Y 0000	0
0	↓	X 0010 Y 0000	1
1	↓	X 0010 Y 0010	2
2	→	X 0010 Y 0010	2
3	↑ →	X 0010 Y 0000	1
4	→	X 0010 Y 0000	1
5	↑ →	X 0000 Y 0000	0
6	↓	X 0000 Y 0000	1
7	→	X 0000 Y 0000	1
8	↓	X 0000 Y 0010	2
9	**hit**		

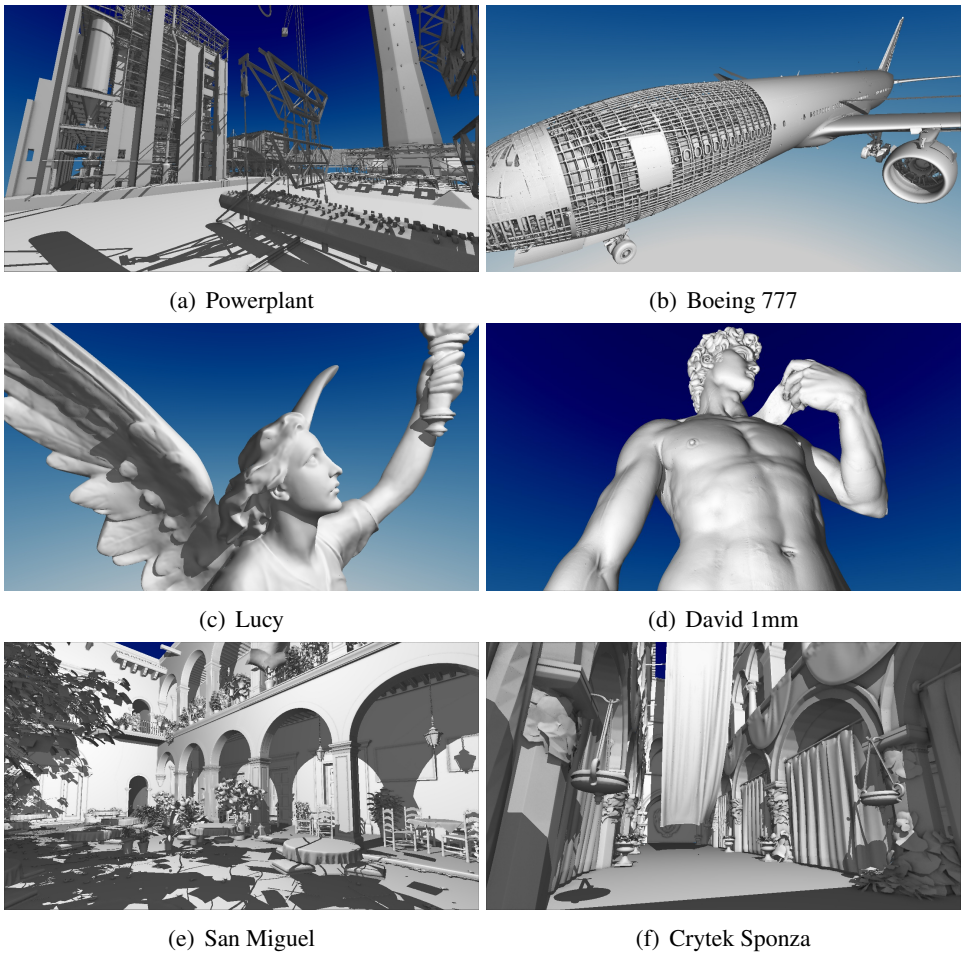
Figure 7. Example of traversal. At left, the example tree, with mirror tags on some pointers. Center, the table of iteration with main actions, current node mirror mask and stack size. At right, a representation of same traversal, with the ray passing though the structure up to find the first intersection.

5.2. Scene rendering

Our implementation uses OpenGL and GLSL. The dataset is fully stored in two texture buffer objects, one for the inner nodes, and one for the leaves. For large datasets that exceed texture buffer objects addressing limits, we use 3D textures.

Sparse voxel DAGs structures are typically used to accelerate tracing for secondary rays, while camera rays are rendered using other structures capable to store normals and material properties (see, e.g., [Kämpe et al. 2013], which uses rasterization for the camera pass). The high compression factor makes it possible, however, to faithfully represent geometry also for primary rays. We have thus implemented a simple deferred shading renderer which uses only our compressed structure to navigate massive models.

In the first pass, the depth buffer is generated by tracing rays from the camera using the algorithm in Listing 1, computing the `proj_factor` parameter of the tracing routing from the camera perspective transformation in order to stop traversal when projected voxel size falls below the prescribed tolerance (1 voxel/pixel in our results). The normals required for shading are then obtained in a second pass by finite differences in the depth buffer using a discontinuity preserving filter. Other shading passes are then performed to compute hard shadows and/or ambient occlusion. While in other settings other structures can be used for storing normals and material properties (see, e.g., [Kämpe et al. 2013; Dado et al. 2016]), this approach also shows a practical way to implement real-time navigation of very large purely geometric scenes from a very compressed representation.



(a) Powerplant (b) Boeing 777

(c) Lucy (d) David 1mm

(e) San Miguel (f) Crytek Sponza

Figure 8. The scenes used in our experiments. All images are interactively rendered using our raytracer from fully GPU-resident data using deferred shading with screen-space normal estimation and hard shadows.

6. Results

An experimental software library, preprocessor and viewer application have been implemented on Linux using C++, OpenGL and GLSL shading language. All the processing and rendering tests have been performed on a Desktop Linux PC equipped with an Intel Core i7-4790K, 32 GB of RAM and an NVIDIA GeForce 980 GTX with 4 GB of video memory.

6.1. Datasets

We have extensively tested our system with a variety of high resolution surface models. Here we present six models which have been selected to cover widely different fields: CAD, 3D scans and video-gaming (see Figure 8). The CAD models, the Powerplant

(12M triangles) and the extremely large and complex Boeing 777 (350M triangles) have been chosen to prove the effectiveness of our method with extremely high resolution datasets with connected interweaving of detailed parts of complex topological structure, thin and curved tubular structures, as well as badly tessellated models that do not always create closed volumes. The 3D Scans Lucy (28M triangles) and Michelangelo's David 1mm (56M triangles), are representatives of dense high resolution scans of man-made objects with small details and smooth surfaces. The fourth and fifth model are the San Miguel dataset (7.8M triangles) and the Crytek Sponza dataset (282K triangles)), which are similar to what can be found on a video-game settings, and, together with Lucy, also provide a direct comparison point with the work on SVDAGs [Kämpe et al. 2013] .

Dataset		$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$
Powerplant	MVoxels	4	17	72	310	1336	5827
	SVDAG Time (s)	0.1	0.4	1.3	3.1	8.6	29.1
	SSVDAG Time (s)	0.3	0.8	2.5	6.3	17.4	52.8
Boeing 777	MVoxels	12	57	268	1242	5699	24633
	SVDAG Time (s)	0.3	1.6	7.2	29.6	121.5	495.71
	SSVDAG Time (s)	1.3	6.0	25.4	107.5	447.1	1846.2
Lucy	MVoxels	6	25	99	395	1580	6321
	SVDAG Time (s)	0.2	0.6	2.3	9.3	37.7	131.7
	SSVDAG Time (s)	0.6	2.1	8.8	37.8	146.2	472.9
David 1mm	MVoxels	4	16	64	257	1029	4116
	SVDAG Time (s)	0.1	0.4	1.7	6.7	25.8	91.2
	SSVDAG Time (s)	0.4	1.5	5.9	25.5	101.2	342.7
San Miguel	MVoxels	12	46	187	750	3007	12045
	SVDAG Time (s)	0.1	0.4	1.6	6.0	19.4	70.4
	SSVDAG Time (s)	0.4	1.4	5.3	19.1	65.8	232.8
Crytek Sponza	MVoxels	40	160	641	2568	10276	41124
	SVDAG Time (s)	0.2	0.7	2.5	10.0	33.5	116.0
	SSVDAG Time (s)	0.6	1.9	6.5	22.5	73.8	226.6

Table 1. Comparison of compression reduction timings. Resolutions are stated in the top row. For each dataset, the first row is the count of non-empty voxels, the second row is the time taken to reduce the SVO to SVDAG, and the third row is the time taken to reduce the SVO to SSVDAG.

6.2. DAG reduction speed

The preprocessor transforms a 3D triangulation into a SVO stored on disk and then compresses it using different strategies. In our out-of-core implementation, the SVO construction using an octree rasterizer can also be interleaved with DAG compression using the multipass strategy.

Preprocessing statistics for the various datasets at different resolutions are reported in Table 1, which indicates the time taken to compress datasets from SVO to SVDAG, as well as to SSVDAG. These times do no include rasterization computation. The

compressor uses OpenMP to parallelize the code, and 8 parallel processes were active in parallel to reduce subtrees. As one can see from the table, the SSVDAG code is slower by factors ranging from $1.8\times$ to $3.8\times$. with respect to SVDAG, which is due to the overhead caused by computing self-similarity.

As a comparison, Crassin et al. [2012] report for the Crytek Sponza dataset a building time of 7.34 ms for the resolution 512^3 on an NVIDIA GTX680 GPU, and Kämpe et al. [Kämpe et al. 2013] report 4.5 s for building an SVDAG from a SVO at resolution $8K^3$ on an Intel Core i7-3930, while the out-of-core scalable voxelizer of Pätzold and Kolb [Pätzold and Kolb 2015] builds a $8K^3$ SVO for the Crytek Sponza dataset in 98 s.

Our conversion times are thus similar to previous node reduction works, and are in any case about faster than the first rasterization step required for creating an out-of-core SVO from the original dataset. In addition, about 4% of the time in our conversion is due to the frequency-based pointer compaction step, which requires a reordering of nodes.

6.3. Compression performance

Table 2 provides detailed information on processing statistics and compression rates of all the test models. We compare our compression results to SVDAG [Kämpe et al. 2013], as well as to the pointerless SVOs [Schnabel and Klein 2006], where each node consumes one byte, a structure that cannot be traversed in random order but useful for off-line storage. For a comparison with ESVO [Laine and Karras 2011], please refer to the original paper on SVDAGs [Kämpe et al. 2013]. It should be noted that the slight differences in number of non-empty nodes in the SVO structure with respect to Kämpe et al. [2013] is due to the different voxelizers used in the conversion from triangle meshes.

Memory consumption obviously depends both on node size and node count. We therefore include for our structure results using uncompressed nodes (USSVDAG in Table 2), with the same encoding employed for SVDAGs [Kämpe et al. 2013], as well as results using our optimized layout (SSVDAG in Table 2). SVDAGs cost 8 to 36 bytes per node, depending on the number of child pointers. Our uncompressed SSVDAGs have the same cost, since symmetry bits are stored in place of padding bytes. On the other hand, our compressed SSVDAGs cost 4 to 34 bytes per node, depending both on the number of child pointers and their size, computed according on the basis of a frequency distribution. In order to assess the relative performance of our different optimizations, we also include results obtained without including symmetry detection but encoding data using our compact representations (ESVDAG).

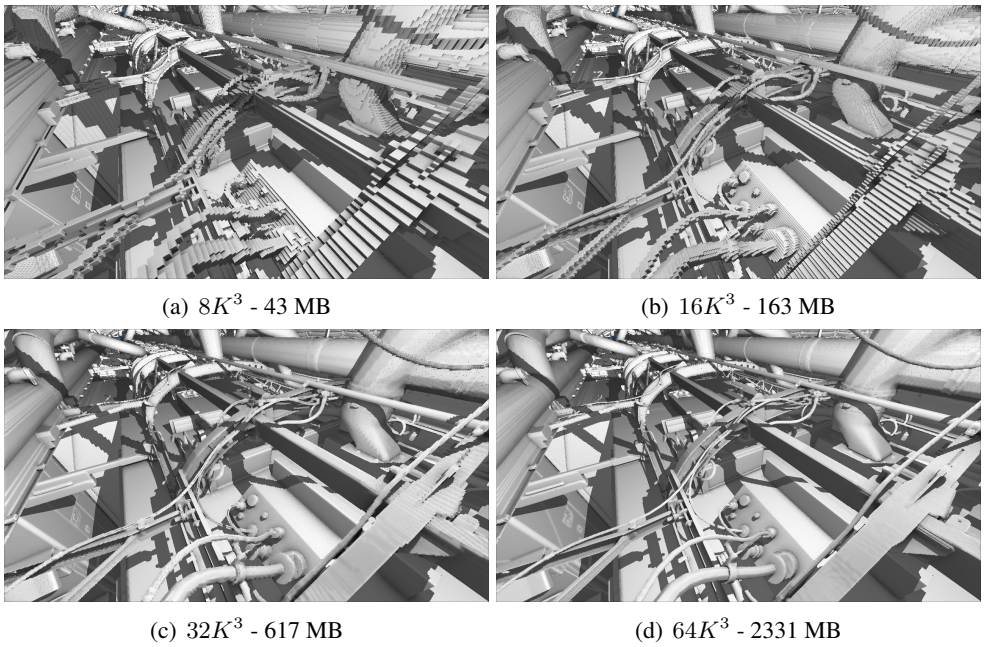
As we can see, all the DAG techniques outperform the pointerless SVO consistently at all but the lower resolutions, even though they offer in addition full traversal capabilities. Moreover, our strategies for node count and node size reduction prove

	Total number of nodes in millions						Memory consumption in MB						bits/vox
Scene	$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$	$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$	$64K^3$
Powerplant (12 MTri)													
SSVDAG	0.1	0.2	0.4	1.0	2.3	5.4	0.7	2.0	5.2	13.3	33.8	85.8	0.123
USSVDAG	"	"	"	"	"	"	1.6	4.0	9.7	23.1	54.9	130.5	0.188
ESVDAG	0.1	0.2	0.5	1.2	2.9	7.0	0.8	2.4	6.3	16.2	41.9	108.6	0.156
SVDAG	"	"	"	"	"	"	1.9	4.9	11.8	28.7	69.9	167.3	0.241
SVO	1.1	5.0	22.0	94.4	404.9	1741.0	1.1	4.8	20.9	90.05	386.2	1660.3	2.390
Boeing 777 (350 MTri)													
SSVDAG	0.3	0.9	3.2	11.3	40.0	140.0	3.0	11.7	43.1	162.9	616.2	2314.4	0.788
USSVDAG	"	"	"	"	"	"	6.9	24.8	83.8	295.0	1042.8	3671.5	1.250
ESVDAG	0.4	1.3	4.3	14.4	48.3	164.4	3.9	15.2	54.0	199.1	731.1	2740.7	0.933
SVDAG	"	"	"	"	"	"	9.2	33.8	112.9	376.7	1260.6	4307.9	1.467
SVO	3.3	15.6	72.8	341.0	1582.8	7282.0	3.1	14.9	69.4	325.2	1509.5	6944.6	2.365
Lucy (28 MTri)													
SSVDAG	0.1	0.4	1.4	4.8	14.4	40.3	1.5	5.3	19.1	67.2	213.6	642.2	0.852
USSVDAG	"	"	"	"	"	"	2.9	9.2	32.3	110.3	332.4	915.5	1.215
ESVDAG	0.2	0.5	1.7	5.7	18.3	52.9	2.1	6.8	24.0	86.5	295.1	896.5	1.190
SVDAG	"	"	"	"	"	"	4.0	11.3	36.8	127.3	419.0	1212.0	1.608
SVO	2.0	8.2	32.9	131.6	526.6	2106.8	2.0	7.8	31.3	125.5	502.2	2009.2	2.666
David 1mm (56 MTri)													
SSVDAG	0.1	0.3	1.1	3.6	11.2	31.8	1.1	3.9	13.6	48.1	159.2	486.7	0.992
USSVDAG	"	"	"	"	"	"	2.2	7.0	23.3	79.7	252.7	716.1	1.459
ESVDAG	0.1	0.4	1.3	4.2	13.8	41.5	1.5	5.0	17.1	61.3	214.3	683.3	1.393
SVDAG	"	"	"	"	"	"	3.0	8.8	27.3	91.8	308.0	938.6	1.913
SVO	1.3	5.4	21.5	85.9	343.2	1372.4	1.3	5.1	20.5	81.9	327.3	1308.8	2.667
San Miguel (7.8 MTri)													
SSVDAG	0.1	0.3	0.9	2.6	7.7	21.6	1.0	3.3	10.3	31.9	98.7	295.9	0.206
USSVDAG	"	"	"	"	"	"	2.1	6.9	21.3	61.8	181.1	509.8	0.355
ESVDAG	0.1	0.3	1.1	3.1	9.1	26.5	1.1	3.7	12.0	37.6	118.7	373.0	0.260
SVDAG	"	"	"	"	"	"	2.3	7.8	24.6	72.0	212.2	621.3	0.433
SVO	3.8	15.3	61.7	248.2	997.8	4004.4	3.6	14.6	58.9	236.7	951.6	3818.9	2.660
Crytek Sponza (282 KTri)													
SSVDAG	0.1	0.4	1.1	2.9	7.6	19.7	1.7	5.2	15.1	42.1	115.7	315.3	0.064
USSVDAG	"	"	"	"	"	"	3.4	9.7	25.8	67.4	172.3	436.8	0.089
ESVDAG	0.2	0.5	1.4	3.7	9.8	25.5	2.1	6.4	18.8	53.6	151.3	417.3	0.085
SVDAG	"	"	"	"	"	"	4.2	11.9	31.8	83.6	218.3	563.5	0.115
SVO	12.8	52.6	212.6	853.9	3421.5	13697.4	12.21	50.2	202.7	814.3	3263.0	13062.9	2.665

Table 2. Comparison of compression performance for various data structures: our Symmetry-aware Sparse Voxel DAG (SSVDAG) is compared with the original sparse voxel DAG (SVDAG), and the pointerless SVO. In order to evaluate the effects of the different optimizations, we also provide results for a version of SSVDAG without pointer compression (USSVDAG) and without symmetry detection (ESVDAG). Resolutions are stated in the top row. On the left, we show the total node count at 2^3 resolution. On the right, we show the total memory consumption of the resulting dataset. The last column states memory consumption per non-empty cubical voxel in the highest built resolutions ($64K^3$).

successful. The USSVDAG structure at $64K^3$ resolution, on average occupies only 79.6% of the storage required by the SVDAG structure, thanks to the equivalent reduction in the number of nodes provided by our similarity matching strategy. An average reduction in size to about 52.4% of the SVDAG encoding is obtained by also applying the frequency-based tagged pointer compaction strategy. Such a strategy is particularly successful since, by matching more pointers, increased opportunities for referencing highly popular nodes arise. This is also proved by the results obtained by the ESVDAG techniques, which uses our data structure but only matches sub-trees if they are equal, as in the original SVDAG. The stronger compression of SSVDAGs makes it possible, for instance, to easily fit all the Boeing model into a 4 GB graphics

board (GeForce GTX 980) at $64K^3$ resolution. Since the Boeing 777 airplane has a length of 63.7 m and a wingspan of 60.9 m, using a $64K^3$ grid permits to represent details with sub-millimetric accuracy (see Figure 9).



(a) $8K^3$ - 43 MB (b) $16K^3$ - 163 MB

(c) $32K^3$ - 617 MB (d) $64K^3$ - 2331 MB

Figure 9. Detail view of the Boeing scene at different resolutions. The compression performance of our method supports real-time rendering from GPU-resident data even at $64K^3$ resolution, while SVDAG memory requirements exceed on-board memory capacity on a 4 GB board (see Table 2).

6.4. Rendering

Since our main contributions target compression, we have focused on verifying the correctness of the different DAG structures, as well as on being able to compare their relative traversal performances, rather than absolute speeds. We have thus implemented a generic shader-based raytracer that shares general traversal code based for the various DAG structures. The different structures are supported by simply specializing the general code through structure-specific versions of the routines that read a node structure, access a child by following a pointer, and applies reflections to 3D indices (see Sec. 5 for details), The SVDAG and USSVDAG version access nodes by fetching data from a GL_R32UI texture buffer object, while SSVDAG uses a GL_R16UI buffer because of the different alignment requirements. 3D textures are used in place of texture buffer objects when the data is so large to exceed buffer addressing limits. This happens only for the Boeing at $64K^3$ resolution for the results presented in this paper. The code does not use any other acceleration or shading structure, and normals

required for shading are generated in screen space. This simple setup also shows that it is possible to use such a terse structure to support interactive navigation of very large models compressed to the GPU.

We have introduced in the renderer two of the most used voxel-rendering optimizations, level-of-detail and the beam optimization introduced by Laine and Karras [Laine and Karras 2011]. For level-of-detail rendering, we simply stop traversal when the projected size of a voxel goes below a threshold (1 voxel/pixel in these results). This is done by computing the `proj_factor` parameter from the camera perspective transformation. For beam optimization, in a coarse rendering pass, we divide the image into 8x8 pixel blocks and cast a distance ray for the corners of these blocks. Traversal is stopped as soon as we encounter a voxel that is not large enough to certainly cover at least one ray in the coarse grid. This is simply done by adapting the `proj_factor` parameter to the coarse image. The subsequent depth pass then conservatively estimates the starting point of each ray from the four neighboring depth values in the coarse grid. Figure 10 illustrates the results obtained during a real-time captured exploration sequence of Crytek Sponza at $64K^3$ resolution. All images are capture at 720p resolution. As one can see, we can trace approximately 100M primary rays/s on such a massive structure even without optimizations. LOD rendering proves dramatically effective in the overall views (boosting the performance to close to 300M rays/s), but, given the fine-grained structure, also provide a non-negligible performance boost even in the close-ups, where beam optimization also becomes effective. Their combination approximately doubles performance in the walkthrough of the inside of the model.

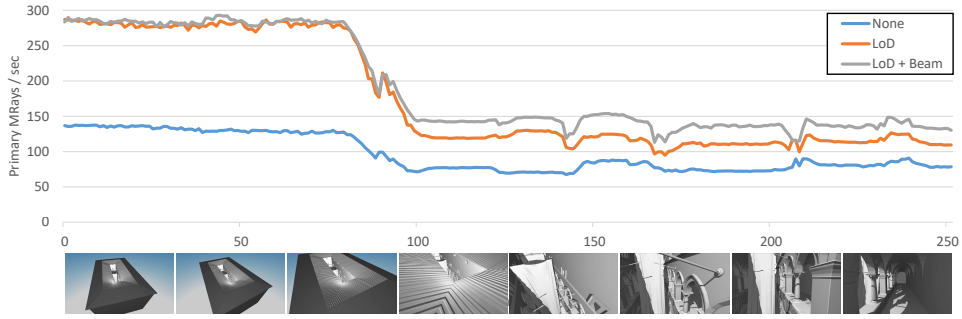


Figure 10. Graph of performance with different optimizations during a real-time captured exploration sequence of Crytek Sponza at $64K^3$ resolution, starting from an overall fly-over (frames 1-80), quickly moving inside the model (frames 80-100), and then performing a walkthrough (frames 100-250).

As illustrated in our accompanying video, we have obtained similar relative performances for all the models. For instance, for the sample viewpoints in Figure 1 of the Powerplant model at $256K^3$ resolution, the views are rendered from farthest to closest (left to right) ranging from 92 to 45 fps for SSVDAG, from 106 to 52 fps

for USSVDAG, and from 107 to 53 fps for SVDAG. All images are rendered in HD (720p) with screen-space normal estimation and hard shadows for one point light. Rendering performance is thus similar for the three implementations, demonstrating that the reduction in memory consumption does not come at the cost of much increased render times.

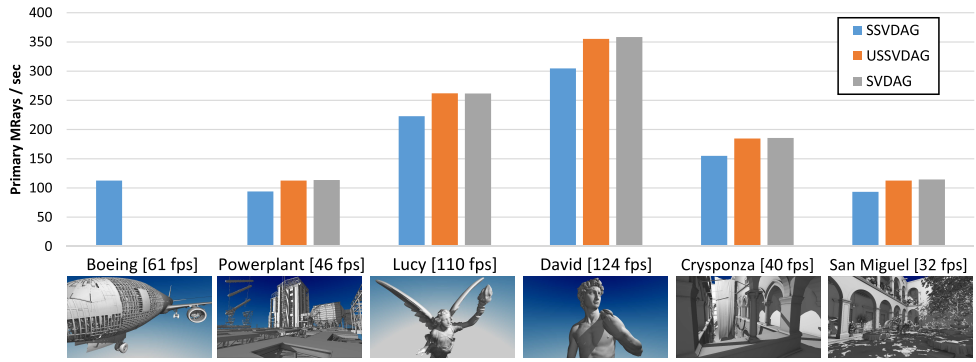


Figure 11. Comparison of primary ray performance for the different structures. Also the frames per second for the final frame (secondary rays, shading, etc.) is shown in each picture. Mirroring is not affecting traversal times, but encoding, even if its performance is still enough for real-time rates.

Figure 11 compares the performance obtained when tracing primary rays for the different datasets at $64K^3$ resolution, for the reference images indicated below the graph. The Boeing 777 dataset does not fit into GPU memory for the other structures, so we only provide results for SSVDAG. As one can see, we can trace from 80M to 300M primary rays/s depending on viewpoint complexity, and the performance of the various structures, as for the Powerplant example discussed above, is consistently similar. It should in particular be noted that reflections impose very little overhead, since USSVDAG is only 1%-2% slower than SVDAG, while variable-rate pointer compression proves a little bit more costly, since SSVDAG has an overhead of 14%-16% with respect to SVDAG. This is probably due to the fact that, while the extra computation required for implementing reflections is well hidden by memory latency, the more elaborate memory layout of pointer compression is more costly. This aspect leaves room for optimization.

Even with our unoptimized shader-based implementation, our SSVDAG structure supports real-time performance for very complex scenes. The Boeing 777 scene can be explored at $64K^3$ resolution in HD (720p) with shading and shadows, at about the same performance as the Powerplant model. Figure 9 shows images taken from the same closeup viewpoint rendered with various voxel resolutions. It is evident how the small voxel dimensions enabled by our compression let appreciate important details that are lost at lower resolutions. The highest resolution is only possible with our SSVDAG and USSVDAG methods, which are the only ones able to fit the entire model

in-core in a 4 GB board.

7. Discussion, Conclusions and Future Work

We have shown that Symmetry-aware Sparse Voxel DAGs (SSVDAGs), an evolution of Sparse Voxel DAGs, allow for an efficient lossless encoding of voxelized geometry representations, in which subtrees that are identical up to a similarity transformations appear only once. Our results demonstrate that this sort of geometric redundancy is common in all tested real-world scenes, ranging from complex CAD models to 3D scans to gaming models, leading to state-of-the-art lossless compression performance. The increased node size with respect to SVOs and SDAGs is quickly balanced by a sizeable reduction in node count. Moreover, pointer overhead is reduced by using fatter leaves and a simple entropy coding scheme. The resulting structure is compact and GPU-friendly, which makes it possible to trace very large scenes while maintaining the visibility acceleration structure fully resident in GPU memory. As the structure can be efficiently constructed out of core, the resulting method is fully applicable to massive data sets, as demonstrated here on large scenes such as the Boeing 777, whose original description exceeds 350M triangles.

As for many data structures and acceleration techniques, our SSVDAG approach has also a number limitations. Handling these limitations indicates interesting areas for future work.

First of all, despite the fact that our compression scheme is lossless, relying on a voxelization scheme leads to a discretization of the original geometry, which, while typically very effective in terms of traversal speed, is not guaranteed to be the most effective in terms of compactness of representation, e.g., for low-poly scenes, or image quality, e.g., for extreme close-up views. These problems are not unique to our method, but typical of all pure voxelization techniques. By increasing compression rates, we significantly improve quality vs. memory costs, allowing for much deeper octrees, but do not eliminate blockiness, which may appear at extreme zoom levels.

A second current limitation of our method is that, while the concept of compression using symmetric DAGs is general, our current implementation is limited to handling only reflective symmetries, and some of the implementation choices explicitly take into account the properties of mirroring transformation (e.g., order independence). While this approach simplifies implementation, it also likely reduces the compression potential, and leaves room for improvement. While the current implementation uses reflections only, an interesting avenue for future work would thus be to investigate other symmetries. A particularly interesting approach would be to evaluate how the method could extend from lossless to lossy compression, by allowing for partial matches of subtrees instead of exact identity up to a transformation. The effect of the errors induced by such approximate techniques should be evaluated in terms of quality/cost

ratio depending on the actual usage, which ranges from primary rays to hard and soft shadows).

Moreover, an additional limitation identified by our benchmarks is that, while the gain in compression rates due to merging symmetric subtrees appear to come at no rendering cost, a tracing overhead of up to 15% appears to be associated to the pointer compaction scheme. This is likely due to the overhead of fetching non-aligned data from GPU memory and to the need to perform bitwise operations for pointer decoding. It should be evaluated whether better layout schemes, or improved shader codes, could reduce this overhead. On the other hand, rearranging nodes based on reference frequency, which is at the core of the pointer compression techniques, has proven very effective, and it would be interesting to evaluate how such rearranging techniques could be further expanded, for example to achieve a better encoding in low-sharing areas.

Finally, similarly to other works on DAG compression [Kämpe et al. 2013; Sintorn et al. 2014; Kämpe et al. 2015], the scheme presented in this paper only supports compression of geometry. While this is acceptable to compute occlusions and for shadowing, general use of the structure would require to map non-geometric properties to voxels (e.g., material or reflectance properties). A promising approach in that respect has been recently presented by Dado et al. [2016] for graphs without symmetries and could be adapted to our technique. An alternative solution would be to totally decouple geometry from material representations, using compressed representations of volumetric textures to overlay a material layer on top of a geometric scene. For maximum compression, it would also be interesting to evaluate how shading normals could be evaluated by differentiating the geometry not only in screen space, but also for secondary rays, expanding techniques used in current volumetric renderers [Beyer et al. 2015].

Acknowledgements

This work is partially supported by the EU FP7 Program under the DIVA project (REA 290277) and by Sardinian Regional Authorities. Datasets are courtesy of the University of North Carolina at Chapel Hill (Powerplant), Dave Kasik and The Boeing Company (Boeing 777), the Stanford 3D Scanning and Digital Michelangelo Repositories (Lucy, David), G. M. Leal Llaguno (San Miguel), and F. Meinl (Crytek Sponza). We thank Ulf Assarsson and Erik Sintorn (Chalmers University) for helpful discussions.

References

AGUS, M., GOBBETTI, E., IGLESIAS GUITIÁN, J. A., AND MARTON, F. 2010. Split-Voxel: A simple discontinuity-preserving voxel representation for volume rendering. In *Proc. Volume Graphics*, IEEE/Eurographics, 21–28. URL: http://www.crs4.it/vic/data/papers/vg2010-split-voxel.pdf. 4

BALSA RODRÍGUEZ, M., GOBBETTI, E., IGLESIAS GUITIÁN, J., MAKHINYA, M., MARTON, F., PAJAROLA, R., AND SUTER, S. 2014. State-of-the-art in compressed GPU-based direct volume rendering. *Computer Graphics Forum 33*, 6 (Sept.), 77–100. URL: http://www.crs4.it/vic/data/papers/cgf2014-star_compressed_gpu_dvr.pdf. 2, 4

BEYER, J., HADWIGER, M., AND PFISTER, H. 2015. State-of-the-art in GPU-based large-scale volume visualization. *Computer Graphics Forum 34*, 8 (Dec.), 13–37. URL: https://doi.org/10.1111/cgf.12605. 6, 26

CRASSIN, C., AND GREEN, S. 2012. *OpenGL Insights*. CRC Press, ch. 22: Octree-Based Sparse Voxelization Using The GPU Hardware Rasterizer. URL: http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf. 6, 20

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '09, ACM, 15–22. URL: https://hal.inria.fr/file/index/docid/345899/filename/CNLE09.pdf. 4, 14

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum 34*, 8, 1921–1930. URL: https://hal.inria.fr/file/index/docid/650173/filename/GIVoxels-pg2011-authors.pdf. 4

DADO, B., KOL, T. R., BAUSZAT, P., THIERY, J.-M., AND EISEMANN, E. 2016. Geometry and attribute compression for voxel scenes. *Computer Graphics Forum (Proc. Eurographics) 35*, 2 (May), 397–407. URL: https://doi.org/10.1111/cgf.12841. 4, 5, 17, 26

FOLEY, T., AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proc. ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, HWWS '05, ACM, 15–22. URL: http://graphics.stanford.edu/papers/gpu_kdtree/kdtree.pdf. 14

GOBBETTI, E., MARTON, F., AND IGLESIAS GUITIÁN, J. A. 2008. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7 (July), 797–806. URL: http://www.crs4.it/vic/data/papers/cgi2008-movr.pdf. 4

HOETZLEIN, R. K. 2016. GVDB: Raytracing sparse voxel database structures on the GPU. In *Proc. ACM SIGGRAPH/Eurographics Symposium on High Performance Graphics*, U. Assarsson and W. Hunt, Eds., HPG '16, Eurographics, 109–117. URL: http://www.ramakarl.com/website/wp-content/uploads/GVDB_HPG2016_CRC.pdf. 5

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive K-D tree GPU raytracing. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '07, ACM, 167–174. URL: http://movement.stanford.edu/papers/i3dkdtree/gpu-kd-i3d.pdf. 14

HUBO, E., MERTENS, T., HABER, T., AND BEKAERT, P. 2008. Self-similarity based compression of point set surfaces with application to ray tracing. *Compututers & Graphics 32*, 2 (Apr.), 221–234. URL: http://doi.org/10.1016/j.cag.2008.01.012. 5

JASPE VILLANUEVA, A., MARTON, F., AND GOBBETTI, E. 2016. SSVDAGs: Symmetry-aware Sparse Voxel DAGs. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '16, ACM, 7–14. URL: http://www.crs4.it/vic/data/papers/i3d2016-symmetry-dags.pdf. 3, 7, 10

KÄMPE, V., SINTORN, E., AND ASSARSSON, U. 2013. High resolution sparse voxel DAGs. *ACM Trans. Graph. (Proc. SIGGRAPH) 32*, 4 (July), 101:1–101:13. URL: http://www.cse.chalmers.se/~uffe/HighResolutionSparseVoxelDAGs.pdf. 2, 3, 4, 5, 12, 13, 14, 16, 17, 19, 20, 26

KÄMPE, V., SINTORN, E., AND ASSARSSON, U. 2015. Fast, memory-efficient construction of voxelized shadows. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '15, ACM, 25–30. URL: http://www.cse.chalmers.se/~d00sint/fastcpvs/fast_cpvs.pdf. 4, 5, 26

KÄMPE, V., RASMUSON, S., BILLETER, M., SINTORN, E., AND ASSARSSON, U. 2016. Exploiting coherence in time-varying voxel data. In *Pro. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '16, ACM, 15–21. URL: http://doi.org/10.1145/2856400.2856413. 5

LAINE, S., AND KARRAS, T. 2011. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (Aug.), 1048–1059. URL: http://doi.org/10.1109/TVCG.2010.240. 2, 3, 4, 5, 6, 14, 16, 20, 23

LEFEBVRE, S., AND HOPPE, H. 2007. Compressed random-access trees for spatially coherent data. In *Proc. Eurographics Conference on Rendering Techniques*, EGSR'07, Eurographics, 339–349. URL: https://hal.inria.fr/file/index/docid/606800/filename/LH07.pdf. 5

PARKER, E., AND UDESHI, T. 2003. Exploiting self-similarity in geometry for voxel based solid modeling. In *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, SM '03, ACM, 157–166. URL: http://doi.acm.org/10.1145/781606.781631. 4

PÄTZOLD, M., AND KOLB, A. 2015. Grid-free out-of-core voxelization to sparse voxel octrees on GPU. In *Proc. ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*, HPG '15, ACM, 95–103. URL: http://doi.acm.org/10.1145/2790060.2790067. 20

SCHNABEL, R., AND KLEIN, R. 2006. Octree-based point-cloud compression. In *Proc. Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG'06, Eurographics, 111–121. URL: http://cg.cs.uni-bonn.de/aigaion2root/attachments/schnabel-2006-octree.pdf. 5, 20

SINTORN, E., KÄMPE, V., OLSSON, O., AND ASSARSSON, U. 2014. Compact precomputed voxelized shadows. *ACM Trans. Graph. (Proc. SIGGRAPH)*

33, 4 (July), 150:1–150:8. URL: http://www.cse.chalmers.se/~d00sint/CompressedShadowsPreprint.pdf. 4, 5, 26

WEBBER, R. E., AND DILLENCOURT, M. B. 1989. Compressing quadtrees via common subtree merging. *Pattern recognition letters 9*, 3, 193–200. URL: http://www.sciencedirect.com/science/article/pii/0167865589900548. 4

Index of Supplemental Materials

Supplementary files for this paper are as follows:

- A video briefly illustrating the method at http://www.jcgt.org/published/0006/02/01/jcgt-ssvdags-demo.mp4.
- Source code of a simplified sample implementation at http://www.jcgt.org/published/0006/02/01/symvox.0.1.zip. This includes code for two applications which have been tested on Windows and Linux:
  - `svbuilder` constructs Sparse Voxel DAGs, with and without symmetries or encoding, as described in this paper.
  - `svviewer` is an OpenGL viewer with GLSL-based traversal.

  See the `readme.txt` file in the zip archive for dependencies, build instructions, and additional details

Author Contact Information

Alberto Jaspe Villanueva
Visual Computing Group
CRS4, POLARIS Ed. 1
09010 Pula (CA) - Italy
ajaspe@crs4.it
http://www.crs4.it/vic/

Fabio Marton
Visual Computing Group
CRS4, POLARIS Ed. 1
09010 Pula (CA) - Italy
marton@crs4.it
http://www.crs4.it/vic/

Enrico Gobbetti
Visual Computing Group
CRS4, POLARIS Ed. 1
09010 Pula (CA) - Italy
gobbetti@crs4.it
http://www.crs4.it/vic/

Alberto Jaspe, Fabio Marton, and Enrico Gobbetti, Symmetry-aware Sparse Voxel DAGs, *Journal of Computer Graphics Techniques (JCGT)*, vol. 6, no. 2, 1–30, 2017
http://jcgt.org/published/0006/02/01/

Received:	2016-08-31		
Recommended:	2016-11-11	Corresponding Editor:	Marc Stamminger
Published:	2017-05-08	Editor-in-Chief:	Marc Olano