High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

GVDB: Raytracing Sparse Voxel Database Structures on the GPU

Rama Karl Hoetzlein[1]

[1]NVIDIA Corporation

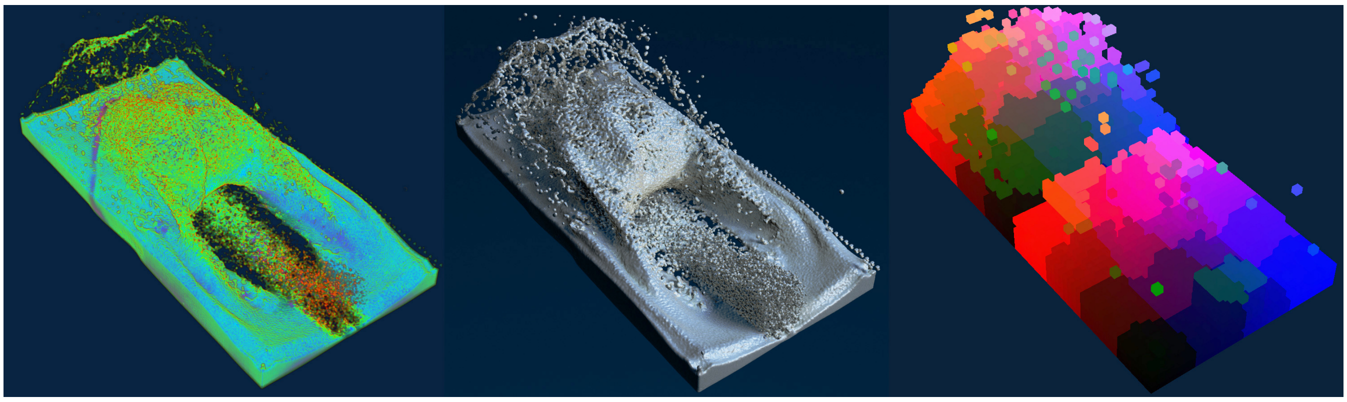

Figure 1: *Volume rendering of a 1 million particle Smoothed Particle Hydrodynamic (SPH) time series simulation with the density field resampled to a 1024^3 sparse GVDB volume, rendered with a) ray sampling with 1 ray/pixel and a custom transfer function at 35 fps, b) multiple scattering and soft shadows with 48 rays/pixel at 1 fps, and c) visualization of empty space skipping showing level-1 bricks.*

Abstract

Simulation and rendering of sparse volumetric data have different constraints and solutions depending on the application area. Generating precise simulations and understanding very large data are problems in scientific visualization, whereas convincing simulations and realistic visuals are challenges in motion pictures. Both require volumes with dynamic topology, very large domains, and efficient high quality rendering. We present the GPU voxel database structure, GVDB, based on the voxel database topology of Museth [Mus13], as a method for efficient GPU-based compute and raytracing on a sparse hierarchy of grids. GVDB introduces an indexed memory pooling design for dynamic topology, and a novel hierarchical traversal for efficient raytracing on the GPU. Examples are provided for ray sampling of volumetric data, rendering of isosurfaces with multiple scattering, and raytracing of level sets. We demonstrate that GVDB can give large performance improvements over CPU methods with identical quality.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

The need for sparse volume data structures to represent uncompressed scalar fields over large domains is driven by the physical sciences and the motion picture industry, where time-varying simulations require volumetric operations with dynamic topology. An ideal data structure would provide features for efficient simulation, dynamic topological changes, sparse compression, fast raytracing and very large addressable spaces. While existing methods excel in a particular area, none are able to combine these advantages to address all goals. We introduce GPU voxel databases, GVDB, as a novel data structure based on a convergent approach that integrates the key features of several techniques to meet these goals while also presenting an efficient framework for simulation and rendering on the GPU.

DOI: 10.2312/hpg.20161197

This work considers the efficient raytracing of dynamic sparse volumes. Many modern data structures are able to achieve efficient rendering of sparse volumes on the GPU. In particular, N^3-trees and tilemaps (or brickmaps) have been found to have branch and traversal characteristics suitable to real time rendering of static data. However, Crassin points out that "animation is a big problem for volume data [rendering]." [CNLE09], which Museth works to address with OpenVDB [Mus13]. While OpenVDB is unique in its consideration of dynamic simulation on "virtually infinite" sparse domains, it achieves these goals through per-voxel iterators. These iterators cache tree traversal pathways for each voxel, which is well suited to multi-core CPU architectures, but not ideal for monolithic kernel execution on single-instruction multiple data (SIMT) architectures since neighbors at non-boundary voxels do not require traversal, leaving many threads idle. There are few approaches that provide the flexibility for very large scale simulations with dynamic topology and efficient kernel execution on modern GPUs.

A key inspiration for the present work is the voxel database structure (VDB), which has found extensive use in motion pictures by addressing several points [Mus13]. The VDB structure focuses on the challenges of simulation with the observation that many desirable tasks require both efficient stencil operations and dynamic changes in topology. These include, for example, algorithms for dilation, flood-filling, advection and diffusion. GVDB does not yet implement simulation but our design anticipates this application. Here we consider raytracing of GVDB data structures with dynamic changes. We approach this problem with a novel memory pooling design that addresses the allocation and deallocation of nodes while retaining the multi-level topology of the VDB structure for large domain simulations.

2. Contributions

GVDB considers the problem of direct single-pass volume raytracing of large, potentially time-varying sparse data to support scattering in a fully integrated raytracing environment. Key contributions of this work are:

- A sparse hierarchical structure for scalar volumetric data
- An efficient memory pooling architecture for dynamic changes in topology
- A novel hierarchical *short-stack* 3DDDA raytracing algorithm entirely on the GPU
- Rendering of ray-sampled volume data with transfer functions, raytracing of isosurfaces, and rendering of level set surfaces
- Major performance improvements over existing methods with identical quality results
- Integration with generic raytracing frameworks such as NVIDIA OptiX for multiple scattering and soft shadows

3. Related Work

Raytracing of volume data is an extensively studied problem. A recent survey of this topic for scientific visualization can be found in Beyer et al. [BHP14], with mention of current GPU techniques. This source also covers recent multi-resolution data structures for sparse data.

Dense volumes can be directly stored and raytraced in 3D texture memory [CN94, CCF94]. This method forms the basis of ray sampling of bricks in sparse methods. Adaptive texture maps by Kraus et al. [KE02] is possibly the first sparse GPU-based volume technique to introduce the separation of index table and a brick texture pool. This approach is employed by others [CNLE09, HBJP12] and by GVDB to store brick data in texture atlases.

Octrees are used as a multi-level sparse volume data structure in several frameworks [GMIG08, CNLE09, KW03]. Krüeger et al. [KW03] use a shallow octree encoded in a hardware texture for empty space skipping. Boada et al. [BNS01] construct octrees to represent level of detail, a common technique for rendering low resolution approximations for non-resident bricks in out-of-core renderers and to approximate voxel antialiasing. We consider level of detail as an extension of our current goals, while our primary effort is to achieve real time rendering of sparse volumes with complete brick data on the GPU. Gobbetti et al. [GMIG08] introduces a stackless approach to raycasting octrees to reduce memory accesses that will be revisited later.

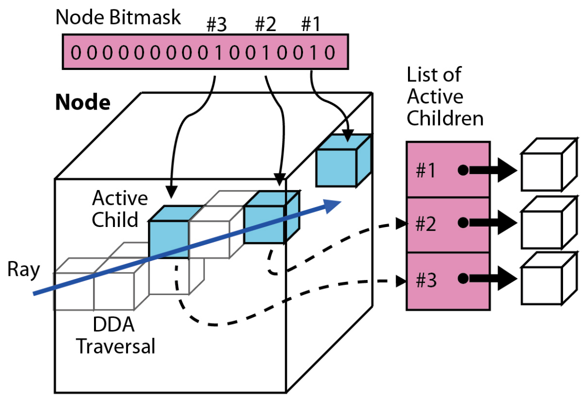

Figure 2: *Basic structure of a VDB Tree. The* node bitmask *indicates which children nodes contain data and are labeled active (bit set to 1). Pointers to the children are stored in the* list of active children*. A ray traverses the node volume using a 3D differential analyzer (DDA) stepping scheme, touching some active children. Others may be active but not touched by the ray (#1). Once an active child is found, the ray may trace into the child node for more detailed data.*

Hierarchical multi-level grids are a recent approach that offer shallow tree depths, large domain sizes, and brick-level sampling at the cost of increased complexity. Efficient GPU raytracing of multi-level grids is an ongoing challenge. Crassin's work on GigaVoxels presents an N^3-tree with level of detail achieved via MIP-mapped 3D bricks at the leaves [CNL08]. Raytracing is accomplished with the *kd-restart* algorithm to resume ray sampling at brick boundaries [HSHH07]. Hadwiger et al. [HBJP12] traces a hierarchical grid by performing brick lookup at each sample while Fogal performs empty space lookup per brick on an index table [FSK13]. We notice that the latter technique is similar to the hierarchical 3D differential analyzer for voxel traversal (H3DDDA) offered by Museth [Mus14] on the CPU, whereby the H3DDDA skips empty space until a brick is found which then initiates ray sampling (Figure 2). This paper examines the adoption of H3DDDA on the GPU.

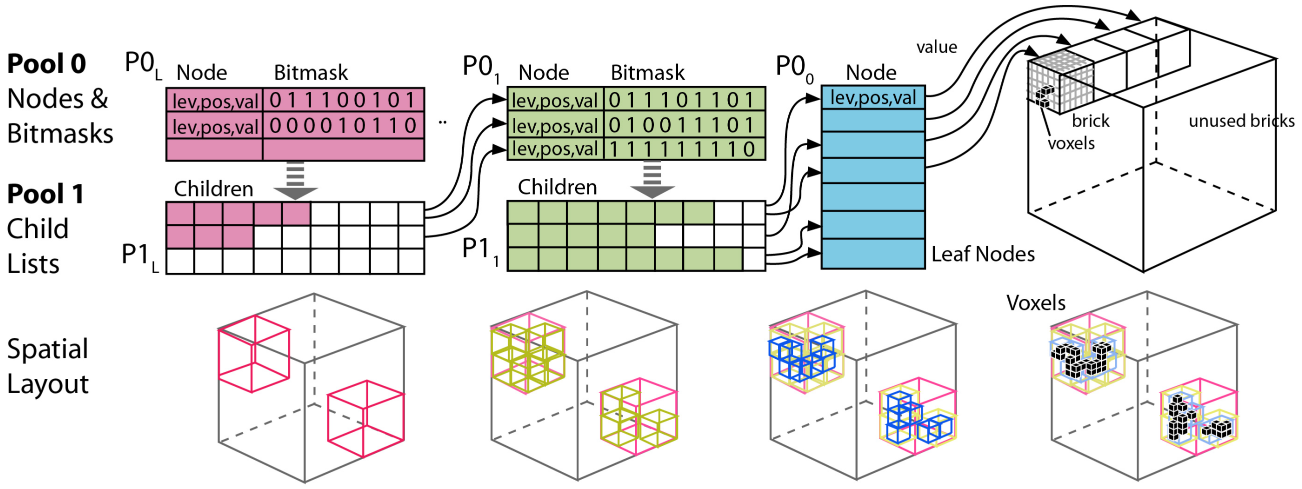
Pool 0
Nodes & Bitmasks
Pool 1
Child Lists
Spatial Layout
P0L
Node
Bitmask
lev,pos,val 0 1 1 1 0 0 1 0 1
lev,pos,val 0 0 0 0 1 0 1 1 0
Children
P1L
P01
lev,pos,val 0 1 1 1 0 1 1 0 1
lev,pos,val 0 1 0 0 1 1 1 0 1
lev,pos,val 1 1 1 1 1 1 1 1 0
P11
P00
lev,pos,val
Leaf Nodes
value
brick
voxels
unused bricks
Voxels

Figure 3: *The GVDB Data Structure uses two memory pools to store a) node data and bitmasks in pool set P0 and b) dynamic lists of children in pool set P1. Nodes are 40 byte headers which contain spatial Position of the node, indices to the Parent and Children lists in other pools, and Values to reference locations in a texture atlas of bricks containing voxels. Bitmasks identify which child nodes are active and bit counting locates a specific child. Multiple atlases may be used for additional data channels or for level of detail reductions at higher levels.*

Recent work with sparse volumes allows for efficient rendering of large meshes and point clouds. Kämpe et al. [KSA13] [KRB*16] recognize that meshes can be efficiently compressed as a directed acyclic graph (DAG) of binary voxels, both in terms of bit-level streaming and through identification of similar bricks, with recent work on temporal meshes. Villanueva et al. [VMG16] extends this by locating graph symmetries. These methods are able to render $64K^3$ scenes on GPU in real time, with less than 0.2 bits/voxel. Reichl et al. [RCSW14] apply binary voxel hashing to accelerate the rendering of point-based fluids. For large scale meshes, sparse voxel octrees are combined with hybrid rasterization to achieve detailed and scalable rendering by Reich et al [RCBW], using orthogonal fragment buffers, and by Chajdas et al. [CRW14], using voxel octrees. These methods rely on sparse binary voxels to accelerate render of meshes and point clouds, whereas we wish to render IEEE floating-point scalar data from original grid-based volumetric sources. We also avoid compression, which would limit performance of dynamic volumes for in situ simulation and rendering applications.

In addition to volumetric ray sampling we also raytrace isosurfaces and level sets on scalar fields. The data structures of Niessner et al. [NZIS13] are relevant as voxel hashing is used to encode signed distance functions (SDF) rather than binary volumes to reconstruct 3D meshes from range data. However, rendering of sparse voxel distance fields is not considered by Niessner as surfaces are extracted for mesh reconstruction. For rendering, Hadwiger et al. [HSS*05] introduces complex shaders on isosurfaces of volumetric data, and Knoll et al. [KWH09] extend this to multi-resolution isosurface rendering. Similarly, the original goal of Museth [Mus13] was to process and render level set surfaces. These tasks are directly applicable to our goals as we wish to render any surface residing on scalar fields.

4. The GVDB Data Structure

4.1. Design

GVDB is a novel data structure for efficient simulation, rendering, and storage of large, uncompressed sparse volumes. The voxel database suffix, VDB, is appropriate since GVDB expresses an identical topological layout in virtual address space as OpenVDB. That is, GVDB retains several similarities to the B+ trees mentioned by Museth [Mus13]. We notice that many advances in modern volumetric GPU-based raytracers are important to consider and develop a novel data structure with a ground up implementation meeting these goals.

Conceptually we define the tree topology in a similar way to Museth [Mus13]. A VDB configuration is a vector which identifies the log_2 resolution of each grid level. For example, the <5,4,3> VDB configuration constructs a 3-level tree with $(2^5)^3 = 32^3$ top level divisions, $(2^4)^3 = 16^3$ mid level divisions, and leaf bricks containing $(2^3)^3 = 8^3$ voxels. An additional level is added to provide a singular root node. Consistent with VDB nomenclature, level 0 is the last value in the vector and defines the leaf level, which allows the tree to arbitrarily increase in depth. A benefit of the VDB specification is that it subsumes several other representations. For example, an <N,N,N,..N> tree defines a N-ary tree, and a configuration of the form <1,1,1,1,1> defines an octree, while a <9,4> tree defines a two-level hierarchy with a 512^3 index atlas and 16^3 bricks.

The design of GVDB consists of a set of nodes at multiple levels in a grid hierarchy, in which each node contains a bitmask and a compacted list of pointers to children. The child list only stores references to active children providing the primary memory savings of VDB. Given a spatial location, if that bit is active, the specific child is located by efficient 64-bit counting on the bitmask to determine the index in the child list. While this slightly increases the cost of lookups it is only performed when a bit is found to be active and saves considerable space compared to indirection tables or octrees with full pointers.

While the concept of a bitmask and child pointers are retained, Figure 2, GVDB has many important design differences from OpenVDB. GVDB uses a single node type instead of the root, interior and leaf nodes of OpenVDB since these have similar functions. For example, leaf nodes in GVDB are interior nodes with active brick values and with an empty child list. Secondly, to provide a convenient way to map CPU and GPU topologies, GVDB introduces integer indexing and a memory allocator to seamlessly transport data between these without address translation. Finally, brick data are stored in GVDB with a 3D texture atlas for efficient hardware trilinear filtering and texture cache access.

4.2. Representation and Layout

The primary contribution of GVDB is an efficient memory layout and data representation for sparse hierarchical grids with dynamic topology. To allow efficient changes in topology the addition and removal of nodes must be fast. Since the number of nodes may exceed 500k for large volumes heap allocation is impractical, which strongly suggests memory pooling. However there are two challenges to memory pooling of VDB nodes. First, the number of voxels can vary on each level, and thus the bitmasks change in size, and second, the list of children for each node can change dynamically.

Memory pooling is achieved with the introduction of two pools for each level of the tree, Figure 3. We notice that while each level has a different resolution, all nodes at a particular level are divided similarly. Thus the bitmasks have identical width for a given level of the tree. We define the first pool group $P0_i$ as a set of pools that exist for tree level i, and have width $W0_i = N + res(i)^3/8$, where N is a fixed width for common node attributes and $res(i)$ is the voxel resolution for that level. For the leaf nodes at level 0, there is no child list so we set $W0_i = N$, keeping in mind that $res(0)$ defines the voxel resolution of a brick in a texture atlas. In this way, each level compactly stores a set of nodes and bitmasks for a given VDB configuration.

A second pool group is needed to store the child lists, and is defined as a set of pools $P1_i$ at each tree level i, with a width containing a list of integer indices of size $W1_i = Kres(i)^3$, where K is a fraction of the maximum number of children. When the child list of a given node overflows, we either dynamically reallocate that pool or move the child list to a larger pool. In practice, we found that the VDB topology is already so compact - requiring less than 1% (average 5MB) of memory compared to the atlas - that K can be set to half or one third of the maximum.

The size of both P0 and P1 pools, the maximum number of nodes they can contain, are initialized with numbers that diminish with level i. When the number of bricks is static and known a priori, we can exactly initialize $P0_0$ and $P1_0$. The higher level pools are initialized at progressively smaller sizes. As references to and from other pools do not change, dynamic reallocation of a specific pool can happen quickly and infrequently.

Given these memory pools, and a brick atlas containing voxel data, the node data is a fixed structure placed in $P0_i$ and containing the following:

Node Attrib	Bytes	Definition
Level	1	Level of the node
Position	12	Position in index space
Value	12	Brick position in texture atlas
Parent	4	Index in $P0_{i+1}$ of the parent
Child List	8	Index in $P1_i$ of the children
Start of mask	$res(i)^3/8$	Bitmask for the node

The node header size (N) is padded to 40 bytes. The largest pool is always the $P0_0$ leaf pool (but has no child lists) so the topology size of a large data set, containing 500k leaf bricks, would average around 20MB. Compare this to a two-level hierarchy using an indirection table with size 512^3 which requires around 500MB assuming 4 byte pointers, or to an octree which would need at least seven levels ($8^6 < 500k < 8^7$).

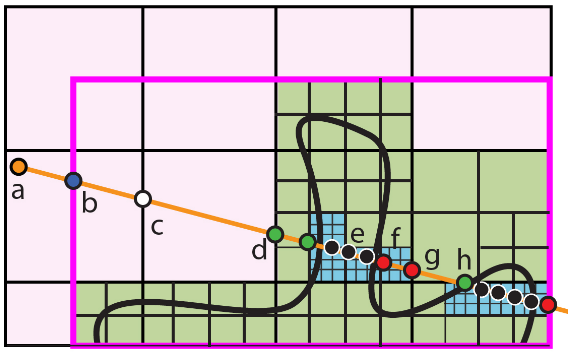

Figure 4: *Raytracing on a hierarchy of grids. a) The ray origin may lie inside the tree, so traversal b) starts at the entry point of the bounding box where c) the node bit is empty, stepping the DDA forward and then d) stepping down the tree and restarting the DDA until the ray e) enters a brick and begins sampling, eventually f) stepping to the next sibling brick, and finally g) stepping out, or h) re-entering the volume again.*

The *Value* attribute is present in all nodes and references a position in the texture atlas for that level. Presently, we only use the level 0 values and a single brick atlas but anticipate that the value index provides an easy way to incorporate level of detail with the addition of downsampled atlases at other levels. We also store an inverse mapping from texture atlas positions to index space for efficient access to spatially neighboring bricks.

The GVDB data structure has several advantages. The entire topology is given by these 2L pools, where L is the maximum tree depth, and can be allocated or transferred to GPU with one call per pool. Additionally, we introduce an accelerated file storage format which writes these pools directly to disk with no translation required. Texture atlases are also written to and read from this file format. For static data that fits in GPU memory this allows sparse volumes to be directly loaded from disk to host mem to GPU mem without a node traversal.

Algorithm 1 Raytracing GVDB Volumes

```
function RaycastGVDB(orig, dir, hit, norm)
    $nodeid[lev] \leftarrow root$
    $node \leftarrow nodeid[lev]$
    $t, tExit[lev] \leftarrow intersectBox(nodeid[lev])$
    $PREPARE\_DDA$                                  ▷ start dda at root
    $p \leftarrow$ local position in brick from DDA
    for $p > 0$ and $p < res[lev]$ do
        $bit \leftarrow (p.z * res[lev] + p.y) * res[lev] + p.x$
        if isBitActive (node, bit) then
            $lev--$                                 ▷ step down tree
            $nodeid[lev] \leftarrow getChild(node, bit)$
            $node \leftarrow nodeid[lev]$
            if lev = 0 then                         ▷ at brick level
                $hit, norm \leftarrow raycastBrick(node, orig, dir)$
                $STEP\_DDA$                          ▷ leave brick and continue
            else
                $tExit[lev] \leftarrow EXIT\_DDA$
                $PREPARE\_DDA$                       ▷ restart dda on entry
            end if
        else
            $STEP\_DDA$                              ▷ bit empty, step dda
        end if
        while $t >= tExit[lev]$ and $lev <= toplevel$ do
            $lev++$                                 ▷ step up tree
            $node \leftarrow getNode(nodeid[lev])$
            $PREPARE\_DDA$                           ▷ restart dda at parent
        end while
    end for
end function
```

4.3. Apron Voxels

When raytracing volumes with trilinear sample interpolation, using gradients in isosurface rendering, or applying compute algorithms that rely on a neighbor stencil, a local neighborhood of voxels is required. This can be problematic at brick boundaries and a common solution is the inclusion of *apron voxels* (or ghost voxels) which is now standard in many brick-based volume renderers [ILC10], with an analysis given in [FSK13]. OpenVDB does not implement apron voxels but instead provides per-voxel cache efficient tree iterators to identify voxels in neighboring bricks. This strategy does not map well to GPU parallelism where we would like interior voxels of a brick to perform the same amount of work per thread as boundary voxels.

GVDB implements apron voxels by increasing the resolution of the texture atlas to provide for their storage. This occurs independently of the topology and involves a change in the value index into the atlas, a function which is built into the brick allocator. GVDB lets the user specify zero or more apron cells during atlas construction while raytracing operations are designed for a specific number. A special compute kernel automatically populates these voxels on-the-fly from a previous time step, similar to Isenburg et al. [ILC10], by performing a neighbor lookup only on the apron voxels using the inverse atlas-to-world mapping described earlier.

5. Raytracing

Our raytracing algorithm is based on observing several constraints. A ray must be able to skip empty space laterally among siblings, jump up and down tree levels, enter and sample bricks, and possibly re-enter the volume multiple times (Figure 4). Hadwiger et al. [HBJP12] resample at each time step, making use of a cache of the current page table hierarchy to improve performance. This has the advantage of simplified ray stepping but avoids gains from large DDA steps. Skipping along DDA node boundaries is achieved by Crassin as the *kd-restart* algorithm traverses entry and exit points while using a top-down traversal on exits [CNLE09]. Museth accomplishes full hierarchical 3DDDA traversal on the CPU with smart iterators [Mus14].

We examine the *kd-restart* algorithm of Foley to develop our own *short stack* raytracer as many of the concepts there can be applied to a hierarchy of grids [FS05]. The key to *kd-restart* is to save the exit point tMax of the current node and advance to that point, restarting tree traversal from the root on each exit. This is suitable in rapidly divided trees such as kd-trees and octrees since there are only two sibling. However, in the spatially subdivided nodes of VDB grids we wish to avoid a *kd-restart* at each entry and exit of a voxel on a given level.

The DDA traversal of GVDB is based on a branchless version of the 3DDDA of Amanatides with the addition of mask and comparison operators instead of branch conditionals for stepping [AW87]. Similar to Museth we implement a hierarchical 3DDDA which steps up and down the tree when an active child is found [Mus14]. Simply unrolling each tree level with multiple sets of DDA variables is ineffective as kernel register pressure becomes a limiting factor. Instead we find that reinitializing the DDA allows us to use only one set of DDA variables at all tree levels.

When traversing down the tree we can restart the DDA at the child node with the current *t*. However, to step up and possibly re-entry a lower level we must save the exit points during traversal. Thus we maintain a *short stack* of exit points during step down traversal and restore these on step up. Our final algorithm, shown in Algorithm 1, is most similar to the *short-stack* method of Horn [HSHH07] for kd-trees, combined with a fast branchless 3DDDA, and bitmask queries to locate active children in the VDB data structure. The algorithm efficiently checks active bits stored in linear memory during DDA steps and never revisits the tree root until the hierarchical DDA traversal finishes. Only when entering a brick is texture memory sampled to accumulate the current pixel color.

6. Results

We examine the performance of GVDB in several ways. First, we implement a similar renderer in OpenVDB with identical output to GVDB to compare the performance of GPU and CPU multi-core rendering. Second, we look at rendering performance and sparse occupancy from the perspective of brick and data size. Finally, we examine the behavior of GVDB as a volume raytracing engine when integrated into generic frameworks for multiple scattering and global illumination.

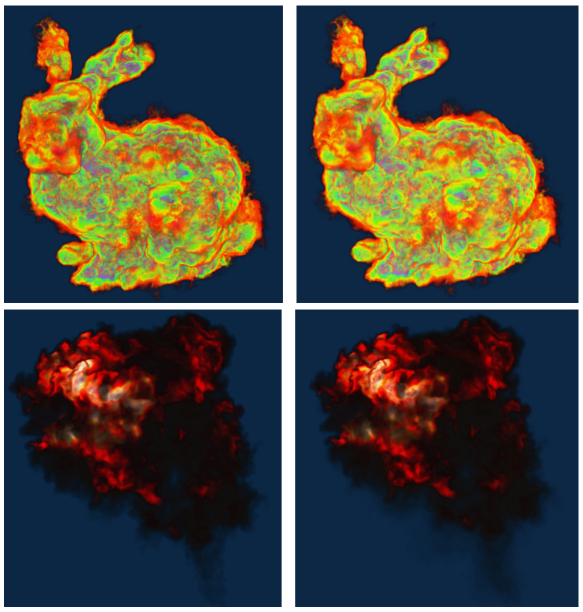

Figure 5: *Comparison of raysampled volumes rendered in GVDB (left) and OpenVDB (right). The bunny volume renders at 34 ms/frame (29 fps) in GVDB and 2235 ms/frame in OpenVDB. The explosion renders at 23 ms/frame (42 fps) in GVDB and 1764 ms/frame in OpenVDB.*

Data Set	Data Res	Occup. (%)	GVDB (ms)	OpenVDB 1 core	OpenVDB 4 core
Explosion	280	39%	23.4	6210	1764
Bunny Cld	584	22%	34.3	4613	2235
Armadillo	1528	2.7%	43.6	1212	281
Bunny	632	7.5%	11.6	845	166
Buddha	1312	9.4%	27.9	723	148

Table 1: *Benchmark results for scenes from OpenVDB loaded into GVDB. All scene are in the <5,4,3> configuration found in the standard OpenVDB package. Notice that level set scenes have significantly lower occupancy as they encode only a narrow band. Data res gives the largest voxel dimension. GVDB measured on a Quadro M6000 and OpenVDB on a 4-core Intel i7-3770K. Scenes are available at http://openvdb.org*

6.1. Performance Comparisons

The raytracing performance of GVDB was measured against multicore CPU-based OpenVDB for standard benchmark scenes. All GPU tests were performed on a Quadro M6000 with 12GB, and CPU tests with a 4-core Intel i7-3770K 3.5ghz with 16GB. Results in Table 1 show that GVDB runs on average 100x-200x faster than single core and 60x faster than 4-core for raysampled volumes. For level sets surfaces the delta is less and GVDB runs 25x-30x faster than single core and 5x-6x faster than 4-core. Renderings in Figure 5 and 6 show that the output is essentially identical in all cases.

Rendering performance was also tested with respect to data and brick size. Using the values in Table 2, we compare these factors in Figure 8 and find a result consistent with the analysis of [FSK13]. Small bricks achieve the best occupancy but place a burden on rendering as the DDA spends more time in entry and exit and less time doing useful work. Large bricks achieve the best rendering performance but at the cost of occupancy. As data resolution increases occupancy is naturally reduced as the discrete volume more closely approximates the data. Assuming larger simulations can reside in GPU memory it becomes more important to improve rendering time with brick dimensions of 64^3 or higher as the data exceeds 2048^3.

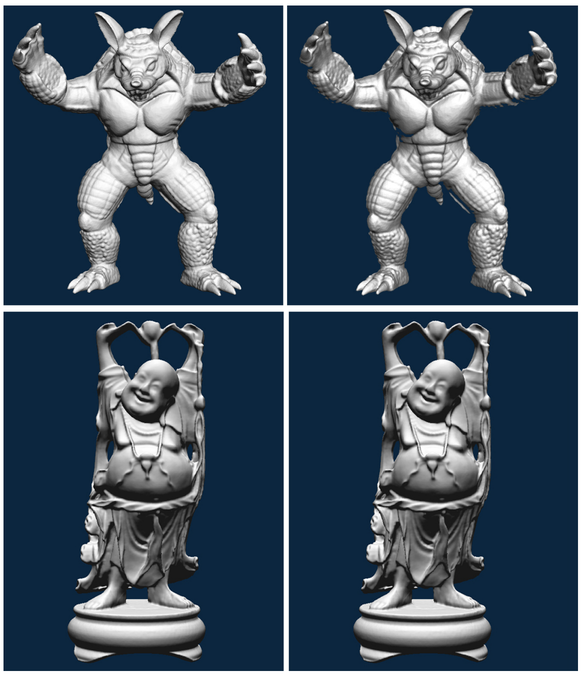

Figure 6: *Comparison of level set surfaces rendered in GVDB (left) and OpenVDB (right). The armadillo renders at 66 ms/frame (15 fps) in GVDB and 281 ms/frame in OpenVDB. The Buddha renders at 24 ms/frame (51 fps) in GVDB and 148 ms/frame in OpenVDB.*

6.2. Multiple Scattering

GVDB was written in CUDA and designed to integrate into generic raytracing solutions such as NVIDIA OptiX [PBD*10]. Two key changes were required. First, ray starts must be possible anywhere including inside the volume itself. This is achieved by initializing to t=0 when bounding box intersection fails. We then provide a ray intersection program for GVDB volumes which returns first hits for isosurfaces and deep colors for ray sampling. These are generically integrated into OptiX to allow inter-reflections between polygons and volumes. For Figure 1 three rays/pixel are cast: one primary ray, one scattering ray, and one low cost shadow ray. These are integrated over multiple frames for sample convergence.

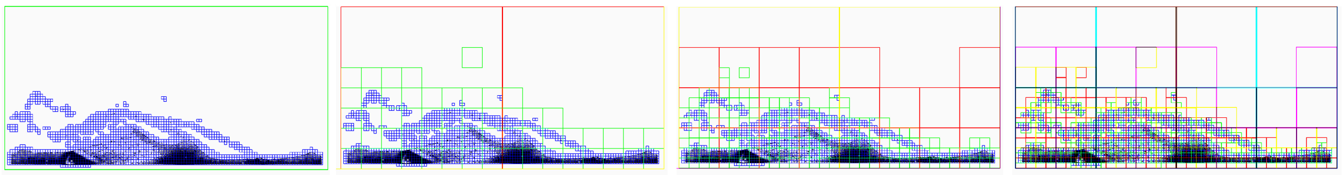

Figure 7: *Example tree topologies studied here. All figures use 16^3 bricks. From left to right, a) tilemaps with depth two and a 128^3 top-level index map, b) an N-ary tree with depth four and 8^3 internal divisions, c) a deeper N-ary tree with depth five and smaller 4^3 divisons, and d) an octree with seven levels. We find that shallow index maps (left image) result in significant memory overhead with excessive DDA traversal, whereas deep octrees (right image) contain too many interior nodes for efficient tree modification. A good balance is achieved with <3,3,3,4> N-ary trees using 16^3 or 32^3 bricks. Coloring of nodes by level is the same in each figure.*

Data Size and Structure				Atlas and Tree Construction *one time cost per data frame*				Rendering *static data with interaction*		
Topology	Data Size (D)	VDB Config	Depth	# Bricks	Occup. (%)	Bricks to GPU (ms)	Tree Build Time (ms)	Empty Skip (ms)	Raycast Total (ms)	fps
N3-tree	512	<3,3,3,3>	4	12,131	27%	2886	9.2	5.8	19.4	51
N3-tree	512	<3,3,3,4>	4	2,036	36%	533	1.6	5.2	17.3	58
N3-tree	512	<3,3,3,5>	4	299	47%	112	0.3	4.9	14.1	71
N3-tree	512	<3,3,3,6>	4	37	58%	49	0.04	4.7	11.5	87
N3-tree	1024	<3,3,3,3>	4	83,218	22%	18940	66.3	7.0	38.8	26
N3-tree	1024	<3,3,3,4>	4	12,131	27%	2947	8.4	5.9	35.4	28
N3-tree	1024	<3,3,3,5>	4	2,036	36%	594	1.6	5.2	28.6	35
N3-tree	1024	<3,3,3,6>	4	299	46%	228	0.3	4.8	22.9	43
N3-tree	1536	<3,3,3,3>	4	267,665	21%	61889	212.1	7.4	64.2	15
N3-tree	1536	<3,3,3,4>	4	38,863	25%	9466	33.3	6.4	52.7	19
N3-tree	1536	<3,3,3,5>	4	5,286	28%	1487	4.4	5.4	52.3	19
N3-tree	1536	<3,3,3,6>	4	945	41%	675	0.9	5.2	37.4	27
N3-tree	2048	<3,3,3,3>	4	616,444	20%	136730	461.0	7.8	92.4	11
N3-tree	2048	<3,3,3,4>	4	83,218	22%	18410	69.8	6.4	64.5	16
N3-tree	2048	<3,3,3,5>	4	12,131	26%	3245	9.3	5.6	57.0	17
N3-tree	2048	<3,3,3,6>	4	2,036	36%	1445	1.8	5.1	49.4	20
Octree	2048	<1,1,..,1,3>	8	616,444	20%	130874	788.0	8.8	125.6	8
Octree	2048	<1,1,..,1,4>	7	83,218	22%	19298	105.5	7.7	108.5	9
Octree	2048	<1,1,..,1,5>	6	12,131	26%	3485	14.1	6.5	93.5	11
Octree	2048	<1,1,..,1,6>	5	2,036	36%	1146	2.7	5.6	87.2	11
N2-tree	2048	<2,2,2,2,4>	5	83,218	22%	18333	72.0	6.6	71.9	14
OpenVDB	2048	<5,4,3>	3	616,444	20%	134381	469.0	13.7	114.2	9
Tilemap	2048	<7,4>	2	83,218	22%	19395	4863.0*	5.4	1202.0*	1*

Table 2: *Performance results for a single frame at 1280x960 from the Waterjet simulation in Figure 1 at different data resolutions and VDB topologies.* Data Size (D) *is the voxel resolution on the longest axis. The volume dimensions are $< D, D11/32, D1/2 >$ voxels. The* Raycast Total *is the time to render a single frame under camera rotation with a single primary ray, volumetric sampling and early ray termination including empty skip time. To estimate an in-situ time-series rendering with changing topology entirely on GPU, add the* Tree Build *time to this. To estimate time-series data streamed from host memory, add the* Bricks to GPU *time to these. * See Section 5.3 regarding Tilemaps*

6.3. Topology and Tree Construction

To measure the performance of GVDB under different data sizes and topologies a time-based sparse volume data set was generated for a one million particle SPH simulation. Direct rendering of SPH data is accomplished with binary voxel acceleration by Reichl et al. [RCSW14]. However, our SPH simulation is used to prepare a synthetic multi-resolution scalar voxel data to emulate in situ grid methods. A liquid SPH simulation was chosen as semi-sparse structure of the fluid allows us to investigate both volumetric raysampling and isosurface rendering. The density field is sparsely resampled to generate volumes at multiple resolutions and brick sizes. Once loaded into GVDB the desired VDB configuration is specified and the corresponding tree topology is constructed for raytracing.

In Table 2 examine a single representative frame (290 of 1200) to understand data scaling. Dynamic topology is investigated by looking at full tree construction time. Tree rebuild is still CPU-based but makes use of our memory pooling structures. Thus, we expect significant improvements from GPU tree construction and

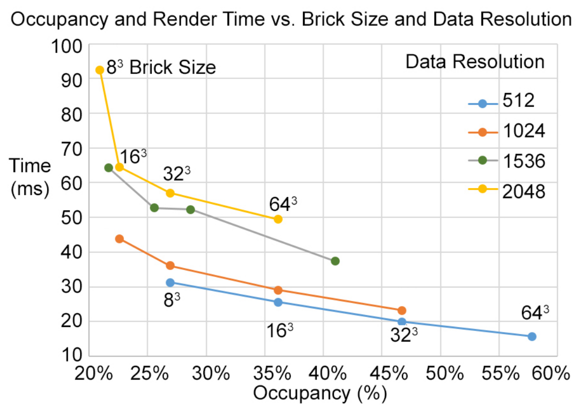

Figure 8: *Occupancy and render time versus brick size and data resolution for a representative frame of the Waterjet SPH simulation. As data scales occupancy is naturally reduced due to tighter discrete approximation while rendering time dramatically increases when the bricks are too small.*

when only incremental changes are made during an in situ simulation. Nonetheless, the results show that in many cases even our current GVDB node pooling tree rebuild performs at interactive rates. Build times in Table 2 are directly correlated with the number of inserted bricks, which is related to the brick dimensions. Larger brick sizes produce a fewer number of bricks resulting in faster tree changes.

An interesting secondary effect on tree build time is the topology choice. Regardless of brick size, octrees behaved on average 30% to 40% slower during node insertion than N_8^3-trees (N-ary trees with 8^3 interior nodes). Since octrees are deeper trees covering the same domain they also contain many more interior nodes and their maintenance during construction is higher. At the opposite end of the spectrum shown in Figure 7, tilemaps are two-level trees using a large index. These also performed slowly which is unexpected since map insertions should be O(1). Unlike octrees having a fast one byte bitmask, for tilemaps this was determined to be bit counting since a 128^3 index contains 2 million voxels or a bitmask with 32768 64-bit words, resulting in a worst case scenario to find or insert nodes in child lists. A direct implementation of brickmaps would achieve O(1) insertions using virtual pointers but with a significant memory cost, while we expect a direct implementation of page table based hashing, as in Niessner et al. [NZIS13], to be competitive in performance.

Overall, these discoveries motivate smaller node dimensions at every level of the GVDB tree and a nice balance is achieved with N_8^3-trees using larger bricks. As shown in Table 2, when data sizes move beyond 2048^3 voxels, 64^3 bricks result in dramatically faster build times of 3 milliseconds or less with only a 10% increase in occupancy.

7. Conclusions

We present GPU voxel databases as an efficient data structure and raytracing technique for sparse volumes with dynamic topology. GVDB rendering performance is significantly faster on the GPU with results visually identical to OpenVDB. While additional performance measures are needed for comparison to GPU-based rendering methods utilizing octrees or page tables over scalar fields, we show that GVDB provides efficient results with many topology configurations. Our next goal is to perform dynamic in situ simulation with GPU-based topology changes. Additional goals include the use of brick compression for faster disk IO and more extensive testing of compute and stencil operations on GVDB volumes. Out-of-core rendering with level of detail is an extension (add a residency bit on each node) that is anticipated in our design but not yet implemented. These future directions explore GVDB as an efficient data structure for in situ simulation and volume rendering with applications to scientific visualization and motion pictures.

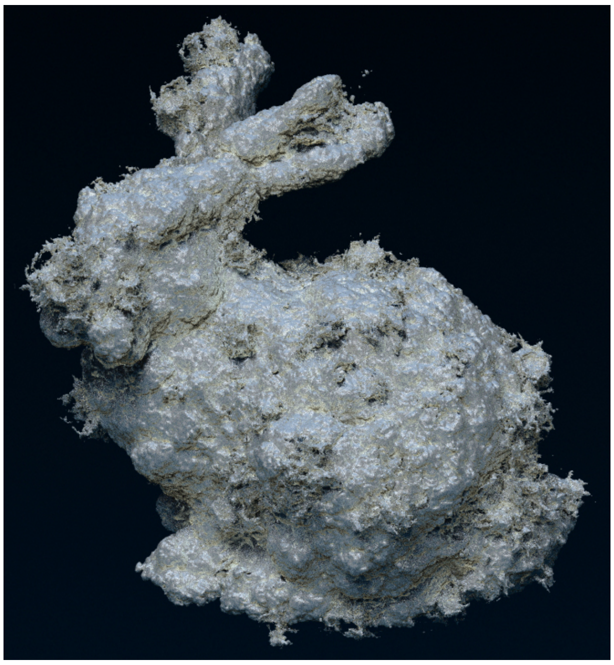

Figure 9: *Raytracing of the Bunny Cloud (584x576x440) dataset with isosurface rendering, multiple scattering and shadows using GVDB embedded in the NVIDIA OptiX raytracing framework. Rays are cast in arbitrary directions without additional preprocessing other than the GVDB sparse volume representation. Hardware trilinear interpolation of the isosurface and gradients for the surface normals are computed on the fly with apron voxels. Rendered in 1.3 seconds at 1280x960 with 16 samples and 3 rays/pixel on a Quadro M6000.*

8. Acknowledgements

Thanks to Tristan Lorach, Holger Kunz and Steven Parker at NVIDIA for their support, and to Tom Fogal, Keith Morley, Dylan Lacewell, Christoph Kubisch and Chris Hebert for feedback. Special thanks to Ken Museth, at Dreamworks Animation and SpaceX, for encouragement and for creating OpenVDB.

References

[AW87] Amanatides J., Woo A.: A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics* (1987), pp. 3–10. 5

[BHP14] Beyer J., Hadwiger M., Pfister H.: A Survey of GPU-Based Large-Scale Volume Visualization. In *EuroVis - STARs* (2014), The Eurographics Association. doi:10.2312/eurovisstar.20141175. 2

[BNS01] Boada I., Navazo I., Scopigno R.: Multiresolution volume visualization with a texture-based octree. *The Visual Computer 17*, 3 (2001), 185–197. doi:10.1007/PL00013406. 2

[CCF94] Cabral B., Cam N., Foran J.: Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings of the 1994 Symposium on Volume Visualization* (New York, NY, USA, 1994), VVS '94, ACM, pp. 91–98. doi:10.1145/197938.197972. 2

[CN94] Cullip T. J., Neumann U.: *Accelerating Volume Reconstruction With 3D Texture Hardware*. Tech. rep., Chapel Hill, NC, USA, 1994. 2

[CNL08] Crassin C., Neyret F., Lefebvre S.: *Interactive GigaVoxels*. Tech. rep., INRIA Technical Report, June 2008. http://hal.inria.fr/docs/00/29/71/63/PDF/rap-rech2-num.pdf. 2

[CNLE09] Crassin C., Neyret F., Lefebvre S., Eisemann E.: Gigavoxels : ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)* (Boston, MA, Etats-Unis, Feb 2009), ACM, ACM Press. 2, 5

[CRW14] Chajdas M. G., Reitinger M., Westermann R.: Scalable rendering for very large meshes. *Journal of WSCG 22* (2014), 77–85. 3

[FS05] Foley T., Sugerman J.: KD-tree Acceleration Structures for a GPU Raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (New York, NY, USA, 2005), HWWS '05, ACM, pp. 15–22. doi:10.1145/1071866.1071869. 5

[FSK13] Fogal T., Schiewe A., Krüger J.: An analysis of scalable GPU-based ray-guided volume rendering. In *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), 2013* (Oct 2013), pp. 43–51. 2, 5, 6

[GMIG08] Gobbetti E., Marton F., Iglesias Guitián J. A.: A Single-pass GPU Ray Casting Framework for Interactive Out-of-core Rendering of Massive Volumetric Datasets. *Visual Computer 24*, 7 (July 2008), 797–806. doi:10.1007/s00371-008-0261-9. 2

[HBJP12] Hadwiger M., Beyer J., Jeong W.-K., Pfister H.: Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach. *IEEE Transactions on Visual Computer Graphics 18*, 12 (2012), 2285–2294. 2, 5

[HSHH07] Horn D. R., Sugerman J., Houston M., Hanrahan P.: Interactive K-d Tree GPU Raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), I3D '07, ACM, pp. 167–174. doi:10.1145/1230100.1230129. 2, 5

[HSS*05] Hadwiger M., Sigg C., Scharsach H., Buehler K., Gross M.: Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum* (2005). doi:10.1111/j.1467-8659.2005.00855.x. 3

[ILC10] Isenburg M., Lindstrom P., Childs H.: Parallel and Streaming Generation of Ghost Data for Structured Grids. *IEEE Computer Graphics and Applications 30*, 3 (May 2010), 32–44. doi:10.1109/MCG.2010.26. 5

[KE02] Kraus M., Ertl T.: Adaptive Texture Maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), HWWS '02, Eurographics Association, pp. 7–15. 2

[KRB*16] Kämpe V., Rasmuson S., Billeter M., Sintorn E., Assarsson U.: Exploiting Coherence in Time-varying Voxel Data. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2016), I3D '16, ACM, pp. 15–21. doi:10.1145/2856400.2856413. 3

[KSA13] Kämpe V., Sintorn E., Assarsson U.: High Resolution Sparse Voxel DAGs. *ACM Transactions on Graphics 32*, 4 (July 2013), 101:1–101:13. doi:10.1145/2461912.2462024. 3

[KW03] Krüger J., Westermann R.: Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (Washington, DC, USA, 2003), VIS '03, IEEE Computer Society, pp. 38–. doi:10.1109/VIS.2003.10001. 2

[KWH09] Knoll A. M., Wald I., Hansen C. D.: Coherent Multiresolution Isosurface Ray Tracing. *Visual Computer 25*, 3 (Feb 2009), 209–225. doi:10.1007/s00371-008-0215-2. 3

[Mus13] Museth K.: VDB: High-resolution Sparse Volumes with Dynamic Topology. *ACM Transactions on Graphics 32*, 3 (July 2013), 27:1–27:22. doi:10.1145/2487228.2487235. 1, 2, 3

[Mus14] Museth K.: Hierarchical Digital Differential Analyzer for Efficient Ray-marching in OpenVDB. In *ACM SIGGRAPH 2014 Talks* (New York, NY, USA, 2014), SIGGRAPH '14, ACM, pp. 40:1–40:1. doi:10.1145/2614106.2614136. 2, 5

[NZIS13] Niessner M., Zollhöfer M., Izadi S., Stamminger M.: Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (TOG)* (2013). 3, 8

[PBD*10] Parker S. G., Bigler J., Dietrich A., Friedrich H., Hoberock J., Luebke D., McAllister D., McGuire M., Morley K., Robison A., Stich M.: Optix: A general purpose ray tracing engine. In *ACM SIGGRAPH 2010 Papers* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 66:1–66:13. doi:10.1145/1833349.1778803. 6

[RCBW] Reichl F., Chajdas M. G., BÃijrger K., Westermann R.: Hybrid Sample-based Surface Rendering. pp. 47–54. doi:10.2312/PE/VMV/VMV12/047-054. 3

[RCSW14] Reichl F., Chajdas M. G., Schneider J., Westermann R.: Interactive rendering of giga-particle fluid simulations. *Proceedings of High Performance Graphics 2014* (2014). 3, 7

[VMG16] Villanueva A. J., Marton F., Gobbetti E.: SSVDAGs: Symmetry-aware Sparse Voxel DAGs. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2016), I3D '16, ACM, pp. 7–14. doi:10.1145/2856400.2856420. 3