Volume 0 (1981), Number 0 pp. 1–8 COMPUTER GRAPHICS *forum*

Out-of-Core Construction of Sparse Voxel Octrees

J. Baert, A. Lagae and Ph. Dutré

Department of Computer Science, KU Leuven, Belgium[†]

Abstract

Voxel-based rendering has recently received significant attention due to its potential in the context of efficiently rendering massively large and highly detailed scenes. Unfortunately, few scenes are available in the form of sparse voxel octrees. In this paper, we present an out-of-core algorithm for constructing a sparse voxel octree from a triangle mesh. Our algorithm allows the input triangle mesh, the output sparse voxel octree, and, most importantly, the intermediate high-resolution 3D voxel grid, to be larger than available memory. We demonstrate that our out-of-core algorithm can construct sparse voxel octrees from triangle meshes using only a fraction of the memory required by an in-core algorithm in roughly the same time, and that our out-of-core algorithm can also handle extremely large triangle meshes.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Graphics data structures and data types

1. Introduction

Voxels have recently received significant attention as a generic representation for geometry and appearance in the context of voxel-based rendering. Examples include the *GigaVoxels* system of Crassin et al. [CNLE09], and the *Efficient Sparse Voxel Octrees* of Laine and Karras [LK10a]. Voxel-based renderers typically operate on high-resolution voxel data stored compactly and hierarchically as sparse voxel octrees (SVO's). Significant advantages of sparse voxel octrees include their regular nature, which makes them easy to work with, and their hierarchical nature, which makes them well suited for level-of-detail. Voxel-based rendering therefore has significant potential in the context of efficiently rendering massively large and highly detailed scenes. Unfortunately, few scenes are available in the form of sparse voxel octrees. Researchers have creatively worked around this limitation, for example by augmenting existing data with details using noise, and by using fractals which can be subdivided infinitely [CNLE09]. The main problem with converting massively large and highly detailed triangle-based scenes into sparse voxel octrees is the size of the intermediate high-resolution voxel data, which typically exceeds the available memory.

In this paper, we present an out-of-core algorithm for constructing a sparse voxel octree from a triangle mesh. Our algorithm allows the input triangle mesh, the output sparse voxel octree, and, most importantly, the intermediate high-resolution 3D voxel grid, to be larger than available memory. The technical contributions of our work are:

- an out-of-core voxelization algorithm that consumes the input triangle mesh and produces an intermediate high-resolution 3D voxel grid in Morton order, using a partitioning scheme for efficient I/O (Sec. 4); and
- an out-of-core sparse voxel octree construction process that consumes the intermediate high-resolution 3D voxel grid in Morton order and produces the output sparse voxel octree, using an optimization for efficient empty node processing (Sec. 5).

We demonstrate that our out-of-core algorithm can construct sparse voxel octrees from triangle meshes using only a fraction of the memory required by an in-core algorithm in roughly the same time, and that our out-of-core algorithm can also handle extremely large triangle meshes.

2. Related work

Sparse voxel octree construction. Several voxel-based rendering systems have recently been proposed, for example, the ray-casting framework of Gobbetti et al. [GMI08], the *GigaVoxels* system of Crassin et al. [CNLE09], and the

---

[†] e-mail: {jeroen.baert, ares.lagae, philip.dutre}@cs.kuleuven.be

Efficient Sparse Voxel Octrees of Laine and Karras [LK10a]. However, these systems mainly focus on efficiently rendering rather than constructing sparse voxel octrees. Gobbetti et al. [GMI08] presented a GPU ray-casting framework for interactive out-of-core rendering of massive volumetric datasets. However, they do not address the construction of the volumetric dataset. Crassin et al. [CNLE09] proposed an approach to efficiently render large volumetric data sets, based on an adaptive data representation depending on the current view and occlusion information. However, they do not explicitly address sparse voxel octree construction. Laine and Karras [LK10a, LK10b] provided an in-depth discussion of various aspects of voxel-based rendering systems, including benefits and drawbacks, storage space requirements, compact data structures, efficient ray-casting algorithms and data management. They proposed a top-down slice-based method for constructing sparse voxel octrees. However, their method is not out-of-core, since the initial build data of the root node can be arbitrarily large. In contrast to the rendering systems above, we do however almost exclusively focus on geometry.

Other tree construction methods. Salmon et al. [SW97] presented a tree construction method in the context of N-body simulation that uses the space-filling Peano-Hilbert curve to build a tree in a top-down fashion. However, their method is designed for an unordered set of points, while our method is designed for a regular voxelization. Kontkanen et al. [KTO11] presented an octree construction method in the context of coherent out-of-core point-based global illumination and ambient occlusion which is somewhat similar to ours. However, their octree construction method is designed for an unorganized set of points with highly varying density.

Voxelization. Over the years, a large number of voxelization methods have been proposed. Representative examples include the methods of Huang et al. [HYFK98], Fang et al. [FC00], Eisemann and Décoret [ED06], Zhang et al. [ZCEP07], Schwarz and Seidel [SS10], and BINVOX [NT03, Min12]. For a recent overview, we refer to the work of Laine [Lai13]. However, these methods typically require a large amount of memory and output an uncondensed 3D voxel grid, while our method is out-of-core and outputs a compact sparse voxel octree. A notable exception is the work of Schwarz and Seidel [SS10], which includes a novel octree-based sparse solid voxelization approach. However, even though the memory requirements are greatly reduced, their method is still limited by the available memory.

Morton order. Morton order [Mor66] (see Fig. 1) is a linearization of an n-dimensional grid that corresponds to the order in which the leaf nodes of the corresponding 2^n-tree (i.e., quadtree in 2D or octree in 3D) are encountered when performing a post-order depth-first traversal of the tree. The Morton code corresponding to a grid cell or a leaf node is its index in Morton order. Morton order is hierarchical: the Morton order for a higher level of the tree corresponds to the Morton order of the coarser version of the grid. Morton order can be obtained by hierarchically subdividing a z-shaped space-filling curve, and is therefore also called z-order. The Morton code of a cell in the n-dimensional grid can easily be obtained from the Cartesian coordinates of the cell using bitwise operations.

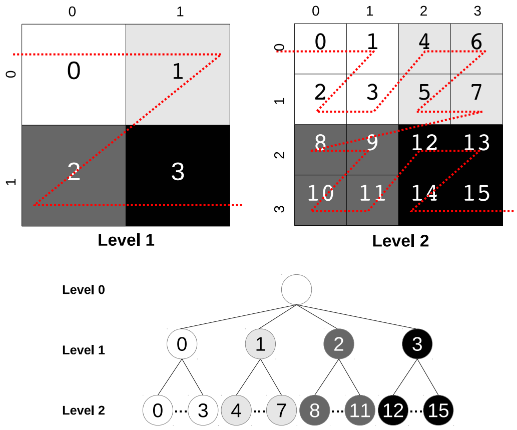

Figure 1: *Morton order (see Sec. 2).*

3. Overview

Our algorithm consists of two processes: (1) the voxelization process and (2) the sparse voxel octree construction process (see Fig. 2). The voxelization process consumes the input triangle mesh and produces an intermediate high-resolution 3D voxel grid. The sparse voxel octree construction process consumes the intermediate high-resolution 3D voxel grid and produces the output sparse voxel octree.

The main insight of our method is that the sparse voxel octree construction process can perform its task in a streaming manner with a space complexity that is logarithmic in the size of the sparse voxel octree *if* the intermediate high-resolution 3D voxel grid is organized in Morton order. This effectively makes the sparse voxel octree construction process out-of-core: The input is consumed and the output is produced in a streaming manner, and the amount of memory required is logarithmic in both input and output. The implication for the voxelization process is that the intermediate high-resolution 3D voxel grid must be produced in Morton order as well.

We will now explain the voxelization process (Sec. 4) and the sparse voxel octree construction process (Sec. 5) in detail. We will explain these processes considering only the geometry of the 3D triangle mesh, we will later briefly consider its appearance (Sec. 6).

4. Voxelization

In this section, we explain the voxelization process of our algorithm in detail. This process consumes the input triangle

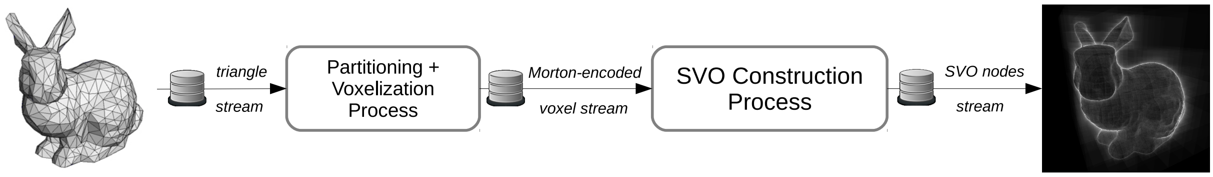

Figure 2: *Overview. Our algorithm consists of a voxelization process, which consumes the input triangle mesh and produces an intermediate high-resolution 3D voxel grid, and a sparse voxel octree construction process, which consumes the intermediate high-resolution 3D voxel grid and produces the output sparse voxel octree (see Sec. 3).*

mesh in a streaming manner and produces the intermediate high-resolution 3D voxel grid in Morton order, which will later be consumed by the sparse voxel octree construction process.

The voxelization process is made out-of-core by partitioning the high-resolution 3D voxel grid into subgrids that fit into the available memory. This process consists of two subprocesses that are executed sequentially: (i) the partitioning subprocess and (ii) the actual voxelization subprocess. The partitioning subprocess partitions the triangle mesh according to the subgrids in a streaming manner. This boils down to testing each triangle against the bounding box of every subgrid. The triangle mesh partitions are temporarily stored on disk. The actual voxelization subprocess sequentially processes the subgrids. For each subgrid, the corresponding triangle mesh partition is consumed in a streaming manner, and the subgrid is output in Morton order. As long as each subgrid corresponds to a contiguous range in Morton order, and the different subgrids are processed in Morton order, the output of this subprocess is the intermediate high-resolution 3D voxel grid in Morton order.

We assume that the bounding box of the triangle mesh is given. If this is not the case, it can be computed in a streaming manner. We determine the size of the subgrids by regularly subdividing the high-resolution 3D voxel grid until a subgrid fits in the available memory. We only use power-of-two resolutions for the high-resolution 3D voxel grid. This method yields subgrids that correspond to contiguous ranges in Morton order. We voxelize triangles using the 6-separability surface voxelization variant of the method of Schwarz and Seidel [SS10]. This method voxelizes triangles independently, which allows the triangles to be processed in a streaming manner. We only output the Morton codes of the non-empty cells of the intermediate high-resolution 3D voxel grid, rather than outputting a boolean value for all of the cells in Morton order, since typically the large majority of the cells in the intermediate high-resolution 3D voxel grid is empty [LD08] (see Tab. 1 and 2).

5. Sparse voxel octree construction

In this section, we explain the sparse voxel octree construction process of our algorithm in detail. This process consumes the intermediate high-resolution 3D voxel grid in Morton order, which was earlier produced by the voxelization process, and produces the output sparse voxel octree.

5.1. Basic version

The key insight for the sparse voxel octree construction process is that Morton order corresponds to a post-order depth-first traversal of the nodes of the octree. This allows us to construct the sparse voxel octree from the intermediate high-resolution 3D voxel grid in a bottom-up manner using a queue of at most 2^d (i.e., 4 in 2D or 8 in 3D) nodes per level of the octree. For example, for a grid resolution of 2048, only 11 queues of 8 nodes are needed.

We provide pseudocode for the sparse voxel octree construction process in Alg. 1.

We illustrate our algorithm by constructing the sparse voxel octree for the 2D voxel grid in Fig. 1.

1. The cells with Morton codes 0–3, which correspond to the level-2 or leaf nodes with the same Morton codes, are read from the input and stored in the queue of level 2.
2. An internal node with level-1 Morton code 0, i.e., the parent of the leaf-nodes with Morton codes 0–3, is created and stored in the queue of level 1, the parent-child relation is recorded, the non-empty nodes in the queue of level 2 are written to the output, and the queue is cleared.
3. The cells with Morton codes 4–7, which correspond to the level-2 or leaf nodes with the same Morton codes, are read from the input and stored in the queue of level 2.
4. An internal node with level-1 Morton code 1, i.e., the parent of the leaf-nodes with Morton codes 4–7, is created and stored in the queue of level 1, the parent-child relation is recorded, the non-empty nodes in the queue of level 2 are written to the output, and the queue is cleared.
5. The cells with Morton codes 8–15 are processed in the same way (see Fig. 3 for a visualization of the queues at this point in the algorithm), after which the queue of level 1 contains the internal nodes with level-1 Morton codes 0–3, and the queue of level 2 is empty.
6. An internal node with level-0 Morton code 0, i.e., the root node, is created and stored in the queue of level 0, the parent-child relation is recorded, the non-empty nodes in the queue of level 1 are written to output, and the queue is cleared.

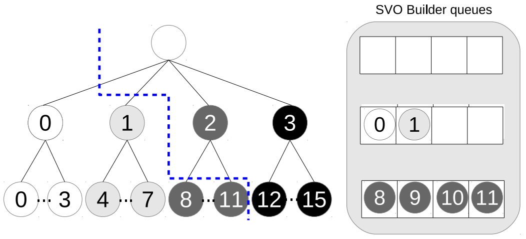

Figure 3: *Sparse voxel octree construction. This is a visualization of the queues associated with each level after all nodes left of the cut have been processed (see Sec. 5.1).*

7. The root node is written to output and the queue of level 0 is cleared.

Note that the voxelization process only outputs the Morton codes of the non-empty cells of the intermediate high-resolution 3D voxel grid (see Sec. 4). In our pseudocode, the input consists of the non-empty cells annotated with their Morton code. Also note that the sparse voxel octree is written to output with a good locality of reference, i.e., siblings are stored next to each other.

Algorithm 1: SVO construction, basic version.

```
1  int current_morton ← 0;
2  while input do
3      // input includes non-empty cells
4      leaf_node l ← consume(input);
5      int nb_empty_nodes ← l.morton - current_morton;
6      while nb_empty_nodes > 0 do
7          queues[$d_{maxdepth}$].push_back(empty_node());
8          flush($d_{maxdepth}$);
9      queues[$d_{maxdepth}$].push_back(l);
10     flush($d_{maxdepth}$);
11     current_morton ← l.morton;
12 // flush full queues upwards from d
13 Function flush(int d)
14     while d > 0 and queues[d].is_full() do
15         internal_node p ← make_internal_node();
16         p.children ← queues[d];
17         write non-empty elements of queues[d] to disk;
           queues[d].clear();
18         queues[$d-1$].push_back(p);
19         d- -;
```

5.2. Optimized version

Although the algorithm explained in Sec. 5.1 works correctly, a lot of time is spent processing empty nodes, since typically the large majority of the cells in the intermediate high-resolution 3D voxel grid is empty.

The key insight for optimizing the basic version of the sparse voxel octree construction process is that pushing back 2^d nodes (i.e., 4 in 2D or 8 in 3D) in a queue is equivalent to pushing back a single node in the queue at the level above, which is much faster. Intuitively, this simply means that directly inserting an empty parent node is more efficient than inserting 2^d empty child nodes and then grouping them.

We provide pseudocode for the optimized version of the sparse voxel octree construction process in Alg. 2.

Although the key insight above is quite simple, applying it over different levels to efficiently insert a large sequence of empty nodes is non-trivial. This is because an empty node cannot be inserted in a queue as long as the queues of lower octree levels still contain nodes. The selection of the queue in which to insert an empty node therefore depends on both (i) the number of empty nodes to insert rounded down to a power of 8 (a in Alg. 2), as well as (ii) the depth of the lowest non-empty queue (b in Alg. 2).

Algorithm 2: SVO construction, optimized version.

```
1  // replaces lines 6-8 in Alg. 1
2  while nb_empty_nodes > 0 do
3      int a ← largest a with $8^a$ < nb_empty_nodes;
4      int b ← largest b with queues[b] is not empty;
5      int $d_i$ ← max($depth - a$, b);
6      queues[$d_i$].push_back(empty_node());
7      flush($d_i$);
8      nb_empty_nodes ← nb_empty_nodes - $8^{depth-d_i}$ ;
```

6. Appearance

Our out-of-core algorithm for constructing a sparse voxel octree from a triangle mesh can easily be extended to handle appearance data as long as this data is available locally when processing the triangle mesh, the intermediate high-resolution 3D voxel grid, and the sparse voxel octree. For example, the voxelization process can easily interpolate vertex attributes such as colors, normals, and texture coordinates, and the sparse voxel octree construction process can easily propagate appearance data such as colors or normals to higher levels in the sparse voxel octree (i.e., to lower levels of detail). However, global data access to e.g. appearance data is inherently incompatible with out-of-core algorithms. This limitation can be alleviated by using a multi-pass approach. For example, the first pass constructs the sparse voxel octree, and the second pass adds texture data to the sparse voxel octree.

7. Implementation

We have implemented our method in C++, which required only approximately 1000 lines of code, including code for I/O and benchmarking. Our reference implementation is available on GitHub (see `https://github.com/Forceflow/ooc_svo_builder`).

We have observed that iterating over a subgrid to output the Morton codes of the non-empty cells (see Sec. 4) is inefficient when the grid is sparse. We optimize this operation by also maintaining a small auxiliary array with Morton codes.

When a cell in the subgrid is marked as non-empty, we also add its Morton code to the array, and when the voxelization is done, we simply output the array rather than iterating over the grid, after sorting it and removing duplicates. We use an auxiliary array that can hold a number of Morton codes equivalent to a small fraction (by default 5%) of the number of cells in the subgrid. In the unlikely case that the array is about to overflow during voxelization, we stop using it and output the Morton codes using the unoptimized version of the operation. Note that this requires to maintain both the auxiliary array as well as the grid. We have not explored other data structures that could potentially be used (e.g., hash tables), since our optimization is simple and effective (see Sec. 8).

We compute Morton codes using a pre-shifted 8-bit lookup table (6 kB) instead of explicitly separating the coordinate bits.

8. Results

In Tab. 1 we show several results of our method. We have obtained the results on a Dell Precision T7500 workstation with an Intel Xeon X5650 CPU and a 7200 RPM SATA II disk.

To compare our out-of-core algorithm with an in-core algorithm, we have applied our method to two different types of triangle meshes: the *David* model, a single object, and the *San Miguel* model, a complete scene. Tab. 1(a) shows the parameters of our method. We have set the resolution of the intermediate high-resolution 3D voxel grid to 2048, which requires 8 GB of memory using a representation of 1 byte per grid cell. We have applied our out-of-core method with 128 MB and 1 GB of available memory, which is respectively 64 and 8 times less memory than an in-core method would require, and using 8 GB of available memory, which is the amount of memory an in-core method would require. Tab. 1(g) shows the total time of our method. Our out-of-core method is roughly as fast as an in-core method, represented by the case with 8 GB of available memory, irrespective of the amount of available memory, but uses up to 64 times less memory. Note that our out-of-core method is sometimes even faster than an in-core method, even when using less memory, since it simply skips subgrids that are completely empty.

To show that our method can also handle extremely large triangle meshes, we have applied our method to the *Atlas* model, a very large triangle mesh consisting of approximately half a billion triangles. We have set the resolution of the intermediate high-resolution 3D voxel grid to 4096, which requires 64 GB of memory using a representation of 1 byte per grid cell. Our out-of-core method can process this 17.42 GB triangle mesh in less than 10 minutes using only 1 GB of memory.

Tab. 1(b) shows the statistics of the input triangle mesh. The triangle mesh is represented using 9 32-bit floats per triangle. Tab. 1(c) shows the statistics of the voxelization process. The timings are broken down into time for I/O input, partitioning or voxelization, I/O output, and miscellaneous operations that do not belong in the previous categories. Tab. 1(d) shows the statistics of the intermediate high-resolution 3D voxel grid. Note that the voxel grid is very sparse. The 8 GB voxel grid is compactly represented using a 64-bit Morton code for every non-empty grid cell. Tab. 1(e) shows the statistics of the sparse voxel octree construction process. The timings are broken down similarly as for the voxelization process. The time for the non-optimized version of the sparse voxel octree construction is also given. Note that our optimized version is over two orders of magnitude faster than the non-optimized version. The sparse voxel octree construction process consumes only roughly 1.6 MB of memory. Tab. 1(f) shows the statistics of the output sparse voxel octree. The sparse voxel octree is represented using 24 bytes per node (a 64-bit reference to the first child of the node, a 1-byte offset for every child of the node, and a 64-bit reference to the data associated with the node). Tab. 1(h) shows a rendering of the output sparse voxel octree using a basic voxel-based ray caster (right side) and a visualization of the output sparse voxel octree obtained by visualizing the number of intersections performed by the ray caster (left side).

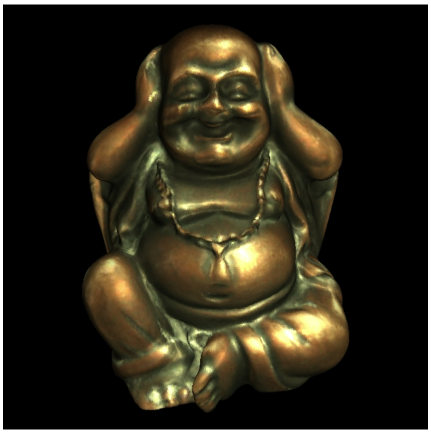

Figure 4: *Appearance. A rendering of a sparse voxel octree of resolution 4096 with appearance data constructed using our out-of-core algorithm (see Sec. 8).*

These results show that our out-of-core algorithm can construct sparse voxel octrees from triangle meshes using only a fraction of the memory required by an in-core algorithm in roughly the same time, and that our out-of-core algorithm can also handle extremely large triangle meshes. In Tab. 2 we show more results.

Our optimization for outputting the Morton codes of the subgrids (see Sec. 7) is highly effective: Compared to a previous version of this work which did not use this optimization [BLD13], the timings improved by roughly a factor of

	David			**San Miguel**			**Atlas**
(a) method parameters							
grid resolution	2048			2048			4096
grid size	8 GB			8 GB			64 GB
available memory	128 MB	1 GB	8 GB	128 MB	1 GB	8 GB	1 GB
(b) input triangle mesh statistics							
# triangles	8.25 M			7.88 M			507 M
size	284 MB			271 MB			17.42 GB
(c) voxelization process statistics							
# partitions / subgrids	64	8	1	64	8	1	64
(c.1) partitioning subprocess statistics							
time, I/O, input	0.22 s	0.22 s	0 s	0.21 s	0.22 s	0 s	235.23 s
time, partitioning	3.07 s	0.47 s	0 s	2.69 s	0.44 s	0 s	155.41 s
time, I/O, output	0.09 s	0.10 s	0 s	0.08 s	0.15 s	0 s	12.77 s
time, misc	0.25 s	0.29 s	0 s	0.29 s	0.29 s	0 s	10.70 s
time, total	3.64 s	1.11 s	0 s	3.23 s	1.06 s	0 s	414.04 s
(c.2) intermediate triangle mesh partitions statistics							
# empty part. / subg.	48	0	0	50	0	0	44
(c.3) voxelization subprocess statistics							
time, I/O, input	0.32 s	0.28 s	0.24 s	0.27 s	0.25 s	0.20 s	69.33 s
time, voxelization	1.90 s	2.65 s	3.60 s	1.99 s	2.98 s	4.15 s	70.54 s
time, misc	0.24 s	0.26 s	0.28 s	0.26 s	0.24 s	0.24 s	16.54 s
time, total	2.47 s	3.21 s	4.12 s	2.52 s	3.48 s	4.60 s	156.42 s
time, total	**6.11 s**	**4.32 s**	**4.12 s**	**5.75 s**	**4.54 s**	**4.60 s**	**570.46 s**
(d) intermediate high-resolution 3D voxel grid statistics							
# non-empty cells	2.89 M			11.0 M			22.9 M
sparseness	99.97%			99.88%			99.99%
size	23 MB			85 MB			174 MB
(e) sparse voxel octree construction process statistics							
time, SVO const.	0.88 s			2.13 s			6.52 s
(unopt. version)	(471.79 s)			(453.25 s)			(N/A)
time, I/O, output	0.56 s			1.38 s			4.36 s
time, misc	0.15 s			0.42 s			1.32 s
time, total	**1.61 s**			**3.95 s**			**12.21 s**
(f) output sparse voxel octree statistics							
# nodes	4.11 M			14.7 M			32.5 M
size	96 MB			338 MB			742 MB
(g) total time	**7.72 s**	**5.93 s**	**5.73 s**	**9.70 s**	**8.49 s**	**8.55 s**	**582.71 s**
(h) visualization and rendering of sparse voxel octree							

Table 1: *Results. Our out-of-core algorithm can construct sparse voxel octrees from triangle meshes using only a fraction of the memory required by an in-core algorithm in roughly the same time (see Sec. 8), and can also handle extremely large triangle meshes.*

	XYZ RGB Dragon			**Nature**			**St. Matthew**
(a) method parameters							
grid resolution	2048			2048			4096
grid size	8 GB			8 GB			64 GB
available memory	128 MB	1 GB	8 GB	128 MB	1 GB	8 GB	1 GB
(b) input triangle mesh statistics							
# triangles	7.2 M			41.3 M			372.76 M
size	248 MB			1.45 GB			13.1 GB
(c) voxelization process statistics							
# partitions / subgrids	64	8	1	64	8	1	64
(c.1) partitioning subprocess statistics							
time, I/O, input	0.21 s	0.17 s	0 s	1.03 s	1.27 s	0 s	100.37 s
time, partitioning	2.88 s	0.54 s	0 s	14.89 s	2.43 s	0 s	129.45 s
time, I/O, output	0.09 s	0.07 s	0 s	1.32 s	0.56 s	0 s	17.54 s
time, misc	0.17 s	0.17 s	0 s	1.31 s	1.29 s	0 s	12.77 s
time, total	3.36 s	0.96 s	0 s	18.56 s	5.56 s	0 s	260.16 s
(c.2) intermediate triangle mesh partitions statistics							
# empty part. / subg.	38	0	0	31	0	0	48
(c.3) voxelization subprocess statistics							
time, I/O, input	0.28 s	0.20 s	0.26 s	1.23 s	1.33 s	1.26 s	10.90 s
time, voxelization	2.02 s	2.72 s	3.74 s	27.56 s	28.00 s	29.67 s	51.31 s
time, misc	0.12 s	0.20 s	0.12 s	1.35 s	1.52 s	1.43 s	12.68 s
time, total	2.43 s	3.13 s	4.13 s	30.15 s	30.85 s	32.37 s	74.89 s
time, total	**5.79 s**	**4.09 s**	**4.13 s**	**48.71 s**	**36.41 s**	**32.37 s**	**335.05 s**
(d) intermediate high-resolution 3D voxel grid statistics							
# non-empty cells	3.9 M			44.17 M			18.60 M
sparseness	99.96%			99.50%			99.80%
size	31 MB			773 MB			145 MB
(e) sparse voxel octree construction process statistics							
time, SVO const.	1.32 s			12.05 s			4.79 s
(unopt. version)	(471.79 s)			(N/A)			(N/A)
time, I/O, output	0.67 s			6.36 s			3.52 s
time, misc	0.25 s			2.17 s			1.23 s
time, total	**2.24 s**			**20.59 s**			**9.56 s**
(f) output sparse voxel octree statistics							
# nodes	5.71 M			59.14 M			26.22 M
size	131 MB			1.38 GB			614 MB
(g) total time	**8.03 s**	**6.33 s**	**6.37 s**	**69.3 s**	**57.00 s**	**52.96 s**	**344.66 s**
(h) visualization and rendering of sparse voxel octree							

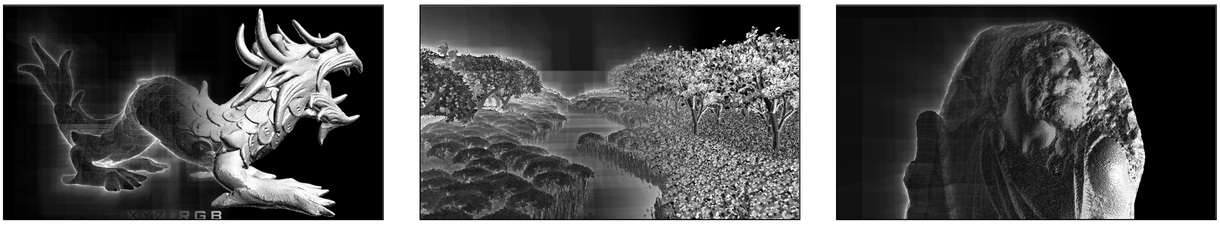

Table 2: *More results.*

two. Of course, the effect of this optimization can vary depending on triangle mesh and hardware architecture.

In Fig. 4 we show a rendering of a sparse voxel octree with appearance data constructed using our method. In this example, we have simply set the voxel color and normal by copying them from the triangle overlapping the voxel.

Our implementation (See Sec. 7) is currently used in the High Fidelity software [Ros13].

9. Comparisons and discussion

Compared to the method of Schwarz and Seidel [SS10], the current state-of-the-art in sparse solid voxelization on the GPU, our method is considerably slower. For example, for the *XYZ RGB Asian Dragon* model (see Tab. 2), our method is roughly two orders of magnitude slower. However, their method is not out-of-core, and is thus limited by the available memory. Also note that their timings do not include I/O from disk to CPU to GPU. In this context, it is important to note that out-of-core methods including ours are often I/O-bound, in which case optimizing the compute time using the GPU does not make much sense, since the overall time is dominated by I/O anyway.

Compared to the CPU method of BINVOX [NT03,Min12], the (to our knowledge) most popular online tool for rasterizing a 3D model file into a binary 3D voxel grid, the voxelization process of our method is roughly an order of magnitude faster. Also note that BINVOX is not an out-of-core method.

The partitioning subprocess is not strictly necessary, the full input triangle mesh could simply be streamed through each subgrid. However, for a large input triangle mesh size and a low amount of available memory, i.e., a large number of subgrids, this will result in a large I/O input time in the voxelization process.

10. Conclusion

We have presented an out-of-core algorithm for constructing a sparse voxel octree from a triangle mesh. We have demonstrated that our out-of-core algorithm can construct sparse voxel octrees from triangle meshes using only a fraction of the memory required by an in-core algorithm in roughly the same time, and that our out-of-core algorithm can also handle extremely large triangle meshes.

Interesting opportunities for future work include investigating whether or not combining out-of-core techniques with GPU methods such as the one of Schwarz and Seidel [SS10] is worthwhile, further reducing the time requires for partitioning and voxelization (e.g., using hierarchical voxelization), extending our method to handle global appearance data (e.g., textures) and (possibly non-linear) filtering of appearance data (see Sec. 6).

Acknowledgements

The *David* and *Atlas* models are courtesy of *The Digital Michelangelo Project*, the *San Miguel* model is courtesy of *Evolucién Visual*, the textured *Buddha* model is courtesy of Kun Zhou. We would like to thank the anonymous reviewers. Jeroen Baert is funded by the Agency for Innovation by Science and Technology in Flanders (IWT). Ares Lagae is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

References

[BLD13] BAERT J., LAGAE A., DUTRÉ P.: Out-of-core construction of sparse voxel octrees. In *Proc. 5th High Perf. Graph. Conf.* (2013), ACM, pp. 27–32. 5

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. 2009 Symp. Inter. 3D Graph. Games* (2009), ACM, pp. 15–22. 1, 2

[ED06] EISEMANN E., DÉCORET X.: Fast scene voxelization and applications. In *Proc. 2006 Symp. Inter. 3D Graph. Games* (2006), ACM, pp. 71–78. 2

[FC00] FANG S., CHEN H.: Hardware accelerated voxelization. *Comp. & Graph. 24* (2000), 433–442. 2

[GMI08] GOBBETTI E., MARTON F., IGLESIAS GUITIÁN J.: A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *Vis. Comp. 24*, 7-9 (2008), 797–806. 1, 2

[HYFK98] HUANG J., YAGEL R., FILIPPOV V., KURZION Y.: An accurate method for voxelizing polygon meshes. In *Proc. 1998 IEEE Symp. Vol. Vis.* (1998), ACM, pp. 119–126. 2

[KTO11] KONTKANEN J., TABELLION E., OVERBECK R. S.: Coherent out-of-core point-based global illumination. *Comp. Graph. Forum 30*, 4 (2011), 1353–1360. 2

[Lai13] LAINE S.: A topological approach to voxelization. *Comp. Graph. Forum 32*, 4 (2013), 77–86. 2

[LD08] LAGAE A., DUTRÉ P.: Compact, fast and robust grids for ray tracing. *Comp. Graph. Forum 27*, 4 (2008), 1235–1244. 3

[LK10a] LAINE S., KARRAS T.: Efficient sparse voxel octrees. In *Proc. 2010 Symp. Inter. 3D Graph. Games* (2010), ACM, pp. 55–63. 1, 2

[LK10b] LAINE S., KARRAS T.: *Efficient Sparse Voxel Octrees – Analysis, Extensions and Implementation*. Tech. Rep. NVR-2010-001, NVIDIA Research, 2010. 2

[Min12] MIN P.: binvox. `http://www.cs.princeton.edu/~min/binvox/`, 2012. 2, 8

[Mor66] MORTON G. M.: *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. Tech. rep., IBM Ltd., 1966. 2

[NT03] NOORUDDIN F. S., TURK G.: Simplification and repair of polygonal models using volumetric techniques. *IEEE Trans. Vis. Comp. Graph. 9*, 2 (2003), 191–205. 2, 8

[Ros13] ROSEDALE P.: The revenge of virtual reality. SIGGRAPH Asia 2013 Keynote Speakers, 2013. 8

[SS10] SCHWARZ M., SEIDEL H.-P.: Fast parallel surface and solid voxelization on GPUs. *ACM Trans. Graph. 29*, 6 (2010), 179:1–179:10. 2, 3, 8

[SW97] SALMON J., WARREN M. S.: Parallel, out-of-core methods for n-body simulation. In *Proc. 8th SIAM Conf. Par. Proc. Sci. Comp.* (1997), SIAM. 2

[ZCEP07] ZHANG L., CHEN W., EBERT D. S., PENG Q.: Conservative voxelization. *Vis. Comp. 23*, 9 (2007), 783–792. 2