

SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering

Markus Hadwiger, Ali K. Al-Awami, Johanna Beyer, Marco Agus, and Hanspeter Pfister

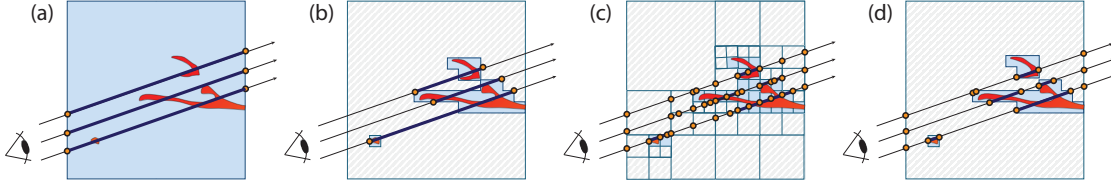


Fig. 1. **Comparison of empty space skipping methods.** Bold rays denote where the volume is sampled; dots depict look-ups for empty space skipping. (a) No empty space skipping: everything is sampled; (b) Standard rasterization of non-empty bounding geometry allows skipping empty space before the first and behind the last non-empty sample, but samples everything in-between. (c) Octree empty space skipping incurs many tree traversal steps in the fragmented space around fine-grained structures. (d) *SparseLeap* skips long segments of empty space even around intricate structures, rasterizing geometry into per-pixel linked lists of ray segments.

Abstract—Recent advances in data acquisition produce volume data of very high resolution and large size, such as terabyte-sized microscopy volumes. These data often contain many fine and intricate structures, which pose huge challenges for volume rendering, and make it particularly important to efficiently skip empty space. This paper addresses two major challenges: (1) The complexity of large volumes containing fine structures often leads to highly fragmented space subdivisions that make empty regions hard to skip efficiently. (2) The classification of space into empty and non-empty regions changes frequently, because the user or the evaluation of an interactive query activate a different set of objects, which makes it unfeasible to pre-compute a well-adapted space subdivision. We describe the novel *SparseLeap* method for efficient empty space skipping in very large volumes, even around fine structures. The main performance characteristic of *SparseLeap* is that it moves the major cost of empty space skipping out of the ray-casting stage. We achieve this via a hybrid strategy that balances the computational load between determining empty ray segments in a rasterization (object-order) stage, and sampling non-empty volume data in the ray-casting (image-order) stage. Before ray-casting, we exploit the fast hardware rasterization of GPUs to create a *ray segment list* for each pixel, which identifies non-empty regions along the ray. The ray-casting stage then leaps over empty space without hierarchy traversal. Ray segment lists are created by rasterizing a set of fine-grained, view-independent bounding boxes. Frame coherence is exploited by re-using the same bounding boxes unless the set of active objects changes. We show that *SparseLeap* scales better to large, sparse data than standard octree empty space skipping.

Index Terms—Empty Space Skipping, Volume Rendering, Segmented Volume Data, Hybrid Image/Object-Order Approaches

1 INTRODUCTION

In volume rendering, a significant part of the computational effort goes into computing a large number of samples from the underlying scalar field. Therefore, one of the most important basic performance optimizations is trying to avoid sampling empty space, i.e., regions where the samples do not contribute to the volume rendering integral. This process is usually called *empty space skipping* or *space leaping* [6].

In recent years, advances in data acquisition techniques have tremendously increased the resolution, size, and complexity of the volume data that need to be visualized. One example are high-resolution microscopy volumes, where the individual voxels can be on the order of micrometers down to a few nanometers in resolution. In neuroscience, for example, the corresponding volume data sets can be multiple to hundreds of terabytes in size [14]. Often, such volumes contain finely detailed structures, such as axons and dendrites of brain tissue [2] or thin blood vessels [31]. However, large volumes containing intricate structures pose a huge challenge to empty space skipping.

Standard empty space skipping techniques, such as those based on octrees, perform well for relatively large connected areas classified as either fully *empty* or fully *non-empty*. However, fine-grained, non-homogeneous regions can lead to severe performance problems when fine structures cause the surrounding space to be subdivided very finely, as depicted in Fig. 1(c). We refer to this as the *fragmentation* of space incurred by a space subdivision such as an octree, which can ultimately lead to very high costs for skipping empty space.

SparseLeap. We propose the novel *SparseLeap* method for efficient empty space skipping in GPU volume rendering. *SparseLeap* is a hybrid object-order (rasterization) / image-order (ray-casting) approach that retains high performance even for large volumes containing fine, intricate structures, because it avoids unnecessary fragmentation of space. Balancing the overall load between object-order and image-order stages allows us to combine the advantages of both worlds.

Our method comprises several key components: First, we introduce an *occupancy histogram tree* that hierarchically tracks three *occupancy classes* for volume regions: *empty*, *non-empty*, and *unknown* (Sec. 5). Second, we describe a traversal algorithm for the occupancy histogram tree that extracts view-independent *occupancy geometry* of nested bounding boxes only where the occupancy class changes (Sec. 6). This significantly reduces the fragmentation of space. Third, we introduce *ray segment lists*, which are per-pixel linked lists of consecutive segments of differing occupancy class. These lists are generated by rasterizing the occupancy geometry, while merging successive segments of the same class, such as several consecutive empty segments, into a single segment (Sec. 7). Finally, empty space skipping during ray-casting is now a simple linear list traversal that skips “as-long-as-possible” empty ray segments without hierarchy traversal.

- Markus Hadwiger, Ali K. Al-Awami, and Marco Agus are with King Abdullah University of Science and Technology (KAUST), Thuwal, 23955-6900, Saudi Arabia. E-mail: {markus.hadwiger, ali.awami, marco.agus}@kaust.edu.sa.
- Johanna Beyer and Hanspeter Pfister are with the John A. Paulson School of Engineering and Applied Sciences at Harvard University, Cambridge, MA, USA. E-mail: {jbeyer, pfister}@seas.harvard.edu.

Manuscript received 31 Mar. 2017; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxx/TVCG.201x.xxxxxxx

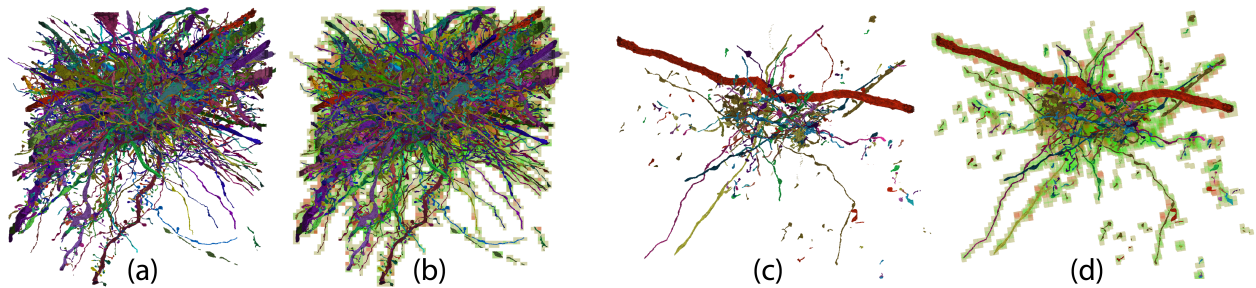


Fig. 2. **Empty space skipping for large, finely detailed volume data.** Large volumes are often sparse and contain thin structures with many individual objects. This leads to highly fragmented empty regions, which makes empty space skipping via tree traversal inefficient because of many small incremental skips. Our *SparseLeap* method enables efficient fine-grained empty space skipping for such sparse volumes with fragmented empty regions using a hybrid object/image-order approach. (a) and (b): more than 4,000 segments in the SEM Mouse Cortex volume; (c) and (d): a subset of less than 100 segments; (a) and (c) show the volume rendering; (b) and (d) also show our occupancy geometry (green and red boxes).

Contributions. In this paper we make several key contributions: (1) A novel method for efficient empty space skipping in large, complex data sets that significantly reduces the fragmentation of space that usually occurs when skipping fine-grained structures in a volume. (2) The design and implementation of efficient algorithms and data structures that combine the advantages of object- and image-order approaches that allow empty space skipping during ray-casting without hierarchy traversal. (3) A performance evaluation of *SparseLeap*, and a comparison to standard octree-based empty space skipping on several large volume data sets with different levels of sparsity.

2 BACKGROUND ON EMPTY SPACE SKIPPING

The goal of empty space skipping is to exclude regions in the volume from sampling that are classified as empty, i.e., regions that are not contributing to the output image. How this can be done depends on the (often hierarchical) volume subdivision employed, as well as on the actual volume rendering method, such as ray-casting or texture slicing, using either a single-pass or a multi-pass rendering approach.

Image-order empty space skipping. State-of-the-art image order volume rendering methods, such as GPU octree ray-casting [3], follow each ray from tree node to tree node. Nodes that are marked as empty can simply be skipped. These decisions are often computed for each ray independently, without exploiting coherence between rays or between successive rendering frames, which is typical for GPU raycasters [5, 11]. Even more importantly, for volumes with thin structures (e.g., the neurites in Fig. 2), octrees subdivide space very finely, and many spatially adjacent nodes will have the same type (empty or non-empty). The corresponding high level of tree refinement forces ray traversal to proceed in unnecessarily small spatial increments. See Fig. 1(c). This large overhead makes empty space skipping inefficient, and significantly reduces the potential performance improvement. Spatial subdivisions that are able to adapt better to thin structures, such as kd-trees, can alleviate these problems, but are rarely used in practice [3], because their subdivision is extremely dependent on the spatial locations of non-empty voxels. If the transfer function or the active set of segmented objects changes frequently, adapting a kd-tree to the new spatial characteristics is not feasible in real time.

Object-order empty space skipping. An alternative to putting almost all of the computational load on the ray traversal stage (e.g., the GPU fragment shader) is to move some of this load into a preceding *object order* stage. In volume rendering, two approaches are common [3]: (1) Multi-pass out-of-core volume rendering often subdivides the volume into relatively large bricks, and each brick is rendered separately [7]. Empty bricks are simply excluded from rendering. This approach is usually restricted to a very coarse granularity, i.e., large bricks, due to the large overhead of multiple rendering passes. (2) Rasterization of approximate bounding geometry allows the ray-casting stage to exclude some empty space [1, 15, 48]. See Fig. 1(b). These approaches can usually only skip “exterior” empty space, or also have to resort to multi-pass rendering and large brick sizes. It is possible to extend these approaches for skipping interior space [40], but this remains unfeasible for hierarchical subdivisions of very large volumes.

The goal of our *SparseLeap* method is to combine the best of these two worlds in a novel hybrid approach that leverages both object-order and image-order components, using fast GPU rasterization.

3 RELATED WORK

The overall volume rendering process and empty space skipping techniques are intimately tied together in most cases. Therefore, we briefly review papers on volume rendering in a comprehensive sense, before focusing more specifically on empty space skipping approaches.

Volume rendering. Recent scalable volume rendering approaches for large data sets typically use GPU ray-casting [5, 11, 14, 15, 23]. In these approaches, large volume data are often represented by an octree, which is then traversed on the GPU using either explicit links between nodes [11], or with octree traversal methods adapted from kd-tree traversal in ray tracing [5, 8]. An important recent development is *ray-guided* volume rendering, which scales with the output image instead of with the input data set size [5, 8, 14]. In our work, we employ a similar ray-guided volume rendering approach. A survey of large-scale volume rendering techniques is given by Beyer et al. [3].

Volume representations. Skipping empty space is often tied to the specific volume representation used. The simplest representation is a regular grid of volume bricks [3]. Kd-trees can adaptively subdivide the volume according to various criteria [41, 43]. Kd-trees are also common for rendering on clusters [7]. However, more often large volumes are represented using octrees [4, 5, 10, 11, 13, 28, 38, 42, 47]. Octrees are less adaptive than kd-trees, but are easier to update after changes to the transfer function or after en/disabling segmented objects. Page table hierarchies are an alternative that has been shown to scale well to very large data [14]. A recent representation for ray-casting voxelized geometry are sparse voxel octrees [25, 26]. Another alternative are adaptive mesh refinement (AMR) hierarchies [18, 19, 46]. Hybrids between grids and trees have also been used [24, 39]. Bounding volume hierarchies (BVHs) are standard in ray tracing, and can be applied to volumes [22], but are not commonly used for regular volume data. Specific representations for sparse volumes can improve storage, performance, or both [24, 35]. The data structure used for empty space skipping is usually intricately linked with the underlying volume representation and rendering method, e.g., standard octree empty space skipping stores non-empty volume data in the same octree, which is traversed for rendering and empty space skipping at the same time. In contrast, the *SparseLeap* data structures for empty space skipping are decoupled from the underlying volume representation.

Empty space skipping. Traditionally, *object-order* vs. *image-order* volume rendering refers to *slicing* vs. *ray-casting* approaches [6]. In the context of empty space skipping, *object-order* mainly refers to the involvement of the *rasterization* of some kind of bounding geometry that approximates the non-empty parts of the volume [1, 40, 48], such as bounding boxes [15] or proxy spheres [30]. If octree or kd-tree nodes are rendered in separate passes, empty space can simply be skipped by excluding the passes of empty nodes [7, 29]. Exploiting ray coherence can improve performance [27], for example by re-projecting the previous frame [21, 44] or occlusion frusta [34].

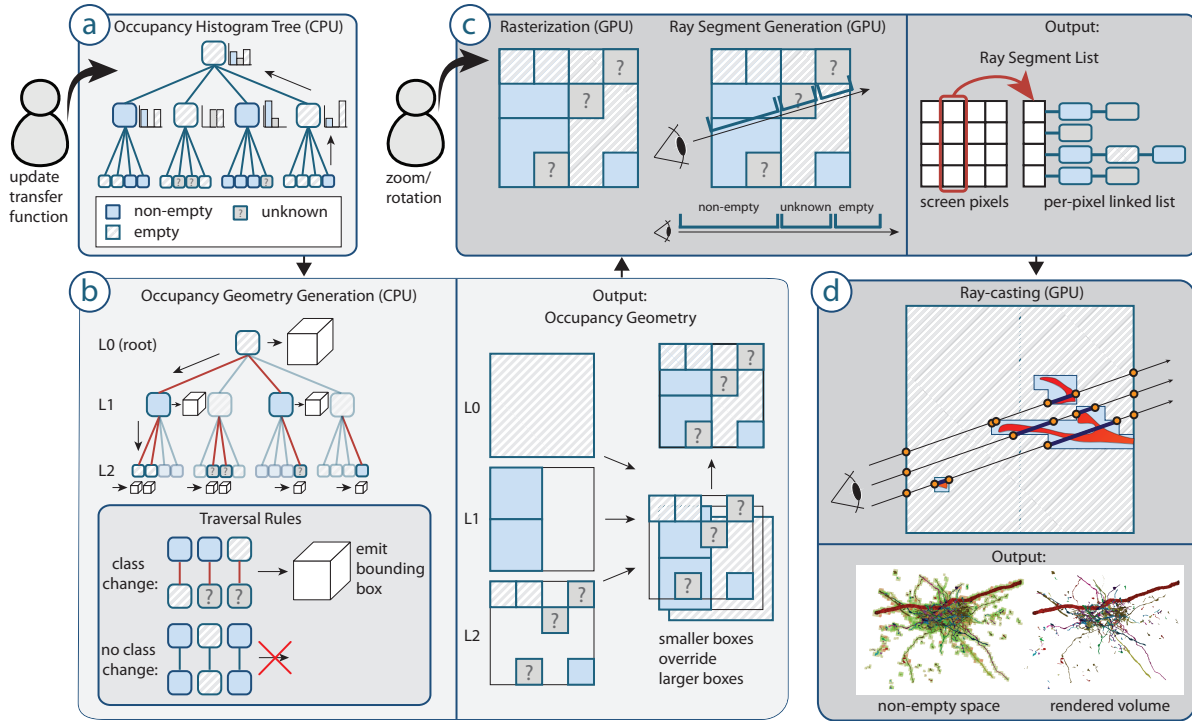


Fig. 3. ***SparseLeap* algorithm overview.** (a) The *occupancy histogram tree* stores hierarchical volume occupancy information, using the classes *empty*, *non-empty*, and *unknown*. (b) Traversal of the occupancy histogram tree creates *occupancy geometry* whenever nested regions differ in occupancy class. The occupancy geometry can be re-used for multiple frames. (c) The occupancy geometry is rasterized into *ray segment lists*, merging successive segments of the same class. (d) Ray-casting leaps over empty space via linear traversal of the ray segment list of each ray.

In the same context, *image-order* mainly refers to GPU ray-casting methods that perform empty space skipping per pixel during the volume sampling step. Typical examples are GPU octree-based ray-casting methods [5, 11] that traverse the octree during ray-casting and skip nodes that are marked as empty. Some work has specifically targeted fibrous structures [33, 36]. It has been shown that AMR hierarchies work better for skipping empty space around thin structures than octrees [18, 19]. However, well-adapted space subdivisions are unfeasible to update interactively in response to transfer function changes or en-/disabling of segmented objects. Strategies for skipping empty space have also been implemented in hardware architectures [32].

SparseLeap is a hybrid method with object- and image-order stages. Hybrid strategies have been employed successfully before, for example in the polygon assisted ray casting (PARC) by Avila et al. [1], and by Westermann and Sevenich [48]. Early approaches often employed graphics hardware rasterization, while performing ray-casting on the CPU. Sobierajski and Avila [40] were able to skip interior empty space instead of only exterior space, by rasterizing a grid of bounding boxes into multiple depth layers, which, although stored very differently, has similarities with our ray segment lists. Bounding geometry can also be represented efficiently, for example using run-length encoding [45].

Geometry and per-pixel linked lists. We take inspiration from occlusion/visibility culling, which is a standard problem in computer graphics that aims at avoiding rendering geometry that is invisible or occluded. Seminal papers are the hierarchical z-buffer [12], and hierarchical occlusion maps [50]. Recent output-sensitive volume rendering architectures [3] refer to this property as ray-guided [5] or visualization-driven [14], ray-casting only non-occluded volume regions. *SparseLeap* employs this strategy to only rasterize geometry where the volume is known to be potentially visible and non-empty.

To create our ray segment lists, we take inspiration from recent order-independent transparency (OIT) methods that use per-pixel linked lists [49]. However, our lists do not need to be sorted after rasterization, which avoids a big performance bottleneck. An alternative is performing rasterization intertwined with volume rendering by doing both in CUDA kernels [20]. The latter approach is very flexible, but we target exploiting the faster hardware rasterization of GPUs.

4 ALGORITHM OVERVIEW

Fig. 3 depicts the *SparseLeap* algorithm and its main data structures. We use an *occupancy histogram tree* to track volume occupancy of regions. Whenever the user enables/disables objects or updates the transfer function, we use the occupancy histogram tree to generate *occupancy geometry*. For each new view, we rasterize this geometry to generate a linear *ray segment list* per ray. Ray-casting then leaps over empty space via linear traversal of the ray segment list of each ray.

Occupancy histogram tree. We track volume occupancy hierarchically, using the three occupancy classes *empty*, *non-empty*, and *unknown*. Each node of the occupancy histogram tree represents a region of the volume. Each leaf node is assigned an occupancy class that is determined via standard culling, i.e., by comparing volume region information, such as min/max scalar value and the set of contained objects, against the current transfer function and the currently active set of objects. Each non-leaf node stores an *occupancy histogram* that is computed by propagating occupancy information up the tree. Each occupancy histogram is the sum of all occupancy histograms below its node. Therefore, each node stores the total count of leaf nodes of the entire corresponding sub-tree in each of the three occupancy classes. Using the non-standard occupancy class *unknown* facilitates lazy (delayed) culling (see Sec. 5.1), which is important for very large data.

Occupancy geometry. From the occupancy histogram tree, we extract a set of *nested* bounding boxes via a linear-time top-down traversal, assigning each box an occupancy class. However, not every tree node “emits” a bounding box. Using a simple set of traversal rules, we emit geometry only when the occupancy class changes while traversing down the tree. This leads to a significant reduction in both geometry and the corresponding fragmentation of space. Corresponding to the spatial extent of nodes, box sizes decrease from the root node down the tree. An important insight of *SparseLeap* is that the occupancy class of a smaller box is able to *override* the class of the larger box that contains it, over the extent of the smaller box. Finally, we exploit the coherence between successive rendering frames by re-using the same occupancy geometry, unless the occupancy itself has changed because the transfer function or the active set of objects has changed.

Ray segment lists. For each new view, the occupancy geometry is rasterized into per-pixel linked lists that store a sequence of successive segments (1D intervals) for each ray. Each segment corresponds to one of the three occupancy classes. During rasterization, consecutive segments of the same class are merged into a single segment. We generate all ray segment lists in parallel by rasterizing the occupancy geometry stored in GPU memory. This approach leverages the extremely fast GPU rasterization hardware, and in this way removes the most significant cost of empty space skipping from the GPU ray-casting shader.

Our method has similarities with GPU order-independent transparency (OIT) approaches using per-pixel linked lists. However, we obtain the ray segment lists in already sorted order, by rasterizing the occupancy geometry accordingly (Sec. 6.2). Therefore, we do not need to sort them afterward, which is a big difference to standard OIT.

Ray-casting. The ray segment lists enable empty space skipping during ray-casting via a simple linear list traversal. The main change to a standard ray-casting implementation is adding an additional outer loop that traverses the ray segment list of the current pixel from front to back along the ray, skipping each encountered *empty* segment. Volume sampling is only performed for *non-empty* segments, as well as for *unknown* segments. Therefore, the *SparseLeap* algorithm can easily be integrated into any existing GPU ray-casting shader.

We now discuss the details of each major component of the *SparseLeap* pipeline just outlined in the following sections.

5 OCCUPANCY HISTOGRAM TREE

The occupancy histogram tree depicted in Fig. 3(a) comprises a spatial subdivision of the volume, i.e., each tree node corresponds to a specific region of space. Our occupancy histogram tree is implemented as an octree, i.e., the spatial region corresponding to each node is an axis-aligned box. However, we emphasize that our occupancy histogram tree serves a different purpose than a volume octree in standard volume rendering. Specifically, the occupancy histogram tree stores a hierarchy of *occupancy histograms* over multiple possible *occupancy classes*. In contrast to standard usage, encoding occupancy information in histograms does *not* mean that every parent node of a *non-empty* node is classified as *non-empty* as well. This is a crucial difference to how occupancy information is propagated in standard volume octrees.

5.1 Occupancy Histograms and Occupancy Classes

An occupancy histogram is a histogram over the three occupancy classes *empty*, *non-empty*, and *unknown*. The histogram stored in a given node of the occupancy histogram tree stores one count for each of the three occupancy classes. Each count corresponds to how many *leaf nodes* of the sub-tree of that node belong to that occupancy class. The leaf nodes of the occupancy histogram tree do not need to store a full histogram. Instead, they directly store the occupancy class corresponding to the spatial region represented by the leaf node.

Occupancy classes *empty* and *non-empty*. For each leaf node, a *culling* step determines whether the spatial region corresponding to the node is *empty* or *non-empty*, by comparing meta-data against the current transfer function and the set of active objects. See below.

Occupancy class *unknown*. In order to be able to scale to very large data, we allow for *delayed* culling, i.e., the lazy determination of the occupancy classes *empty* or *non-empty*. This is facilitated by the class *unknown*, which denotes that it is not yet known whether the spatial region corresponding to a leaf node is actually *empty* or *non-empty*. To fully support this, our ray-caster generates an *occupancy miss* when it encounters a ray segment of class *unknown* (Sec. 8.2).

5.2 Tree Updates

In order to determine the occupancy class of a given leaf node of the occupancy histogram tree, actual *culling* must be performed for that node. Apart from allowing delayed culling (see below), we do this in the standard way, i.e., by comparing stored *meta-data* of the leaf node against global user-determined settings, such as the current transfer function, and the current set of active (enabled) segmented objects:

- **Segmented objects:** Culling requires meta-data of the set of objects contained in each node. If all contained objects are globally disabled, the occupancy class of the node will be set to *empty*. Otherwise, it will be set depending on the transfer function.
- **Transfer function:** Culling requires meta-data of min/max scalar value of each node. For completely transparent (zero opacity) nodes, this will result in an occupancy class of *empty*, otherwise in a class of *non-empty*. Iso-surfaces can be culled analogously.

Delayed culling. Culling can be performed in a delayed fashion, where the occupancy class of a leaf node is initially set to *unknown*, and only changed to *empty* or *non-empty* after the (asynchronous) culling process has completed. In addition to delayed culling, our implementation supports even computing the meta-data required by culling in a delayed fashion. In order to support this, the meta-data are allowed to be missing. When an occupancy miss is reported for a node that is still missing its meta-data, it will be computed at that time.

Occupancy histogram propagation. The occupancy histograms are computed by propagating the occupancy class information of leaf nodes up the tree. In order to do this, we traverse the occupancy histogram tree recursively in *depth first order*, starting at the root: Each non-leaf node first descends to its child nodes, and, after the child traversal returns, computes its occupancy histogram as the sum of the occupancy histograms of its child nodes, which are now correctly set.

Growing and pruning the tree. When an occupancy miss is reported that corresponds to a node that does not exist in the occupancy histogram tree, the tree is automatically grown to include that node. The initial occupancy class of new nodes is always set to *unknown*. It will be changed to *empty* or *non-empty* after culling has been performed. We automatically retire unused (invisible) tree nodes, i.e., nodes that are occluded or outside the view frustum. Doing so makes our approach output-sensitive, i.e., it scales with the visible output instead of the whole volume. This is done by deleting nodes, and possibly their entire sub-tree, using a least-recently used strategy (Sec. 8.3).

6 OCCUPANCY GEOMETRY GENERATION

From the occupancy histogram tree, the corresponding occupancy geometry illustrated in Fig. 4 is extracted by traversing the tree using simple traversal rules that determine whether a node’s bounding box should be emitted or not. Note that this process is *view-independent*.

6.1 Occupancy Histogram Tree Traversal

To generate the occupancy geometry, we traverse the occupancy histogram tree in *breadth first order*, visiting child nodes in a fixed order (not in visibility order). The resulting occupancy geometry therefore always stores larger boxes (coarser tree levels) before smaller boxes (finer tree levels), which optionally allows stopping the traversal when a given budget for the number of boxes has been reached.

According to the traversal rules below, a given node might simply be skipped during traversal, or it might emit the geometry of the node’s bounding box (we only store box center and size), together with the node’s occupancy class, as well as the occupancy class of its *parent*.

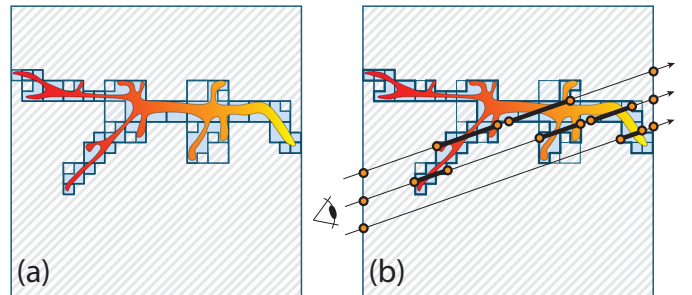


Fig. 4. **Occupancy geometry generation and ray segments.** (a) The occupancy geometry comprises view-independent nested bounding boxes of different occupancy class, reducing the fragmentation of space. (b) Consecutive segments of the same occupancy class are merged during rasterization of the occupancy geometry into ray segment lists.

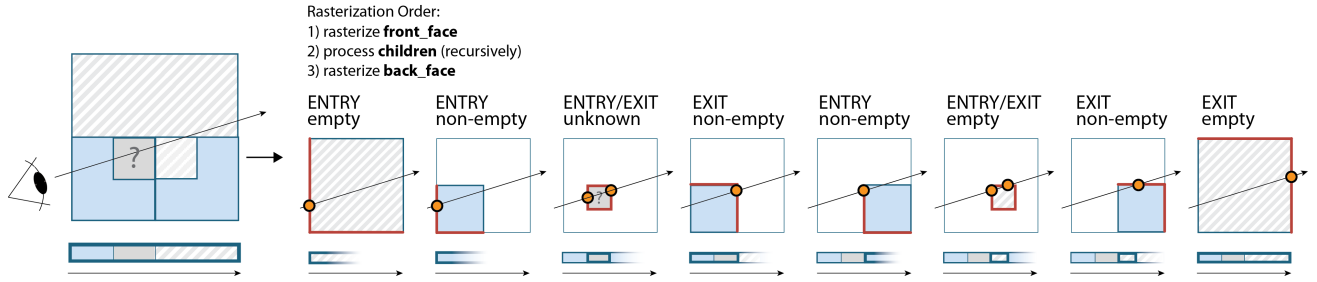


Fig. 5. **Ray events and ray segment list generation.** Ray segment lists are generated by rasterizing *nested* occupancy geometry bounding boxes. The rasterization of their front- and back-faces, respectively, results in *ray events* at the corresponding intersection positions: *entry* events for front faces, and *exit* events for back faces. After all ray events have been rasterized (right), including possibly merging or deleting events, the result is the final ray segment list (left): a sequence of ray segments (intervals between consecutive ray events), with one occupancy class per segment: *empty*, *non-empty*, or *unknown*. Left: Occupancy geometry and corresponding output ray segment list. Right: Step-by-step generation.

Traversal rules. The occupancy histogram tree produces an array of bounding boxes with occupancy classes using the following rules:

- The tree is traversed in breadth first order starting at the root. Each node is passed in the *occupancy class* of its parent node. The parent class for the root node is *invalid*, defined as not equal to any other class. Therefore, the root node always emits its bounding box (see below), i.e., the whole volume bounding box.
- Each node determines its occupancy class. For *leaf nodes* the occupancy class is already stored in the node and does not have to be computed. Each visited *non-leaf node* inspects its occupancy histogram to determine its own occupancy class. The class is set to the *majority vote* over the occupancy histogram, i.e., the class that occurs the most often in all leaf nodes of its sub-tree.
- *Geometry emission:* Each visited node compares its occupancy class with the class of its parent: If the class is the same, no bounding box is emitted. If the class is different, a bounding box with the respective occupancy class info is emitted for the node.
- If the already emitted number of bounding boxes exceeds a pre-specified (optional) resource limit, traversal is stopped.

Occupancy geometry and fragmentation of space. Fig. 4(a) illustrates an example occupancy geometry of nested bounding boxes resulting from these traversal rules. Fig. 2(b, d) depicts actual occupancy geometry as green (*non-empty*) and red (*empty*) boxes. Using our approach, space is significantly less fragmented than when a standard octree is used. However, we observe that some fragmentation of space still remains. These cases are resolved during the subsequent rasterization of ray segment lists, where consecutive ray segments of the same occupancy class are detected and merged (Fig. 5). The overall resulting depth complexity is very low, which is visualized in Fig. 7 (left).

6.2 Occupancy Visibility-Ordering

The occupancy geometry itself is view-independent. However, the rasterization of ray segment lists below must be performed in visibility order with respect to the current view point. Therefore, whenever the view changes, we compute a visibility-sorted index array that references the current occupancy geometry in front-to-back visibility order.

Visibility traversal. We traverse the occupancy histogram tree in *depth first order*, visiting child nodes in front-to-back visibility order. In contrast to a standard tree traversal in visibility order, we do not only emit indices for leaf nodes, but for all non-leaf nodes as well.

In fact, each node index is emitted *twice*, such that the subsequent rasterization is able to differentiate bounding box front and back faces, respectively (in the fragment shader). Each non-leaf index is emitted *before* its child nodes are visited, and is again emitted *afterward*. The former index is marked for *front face* rasterization. The latter index is marked for *back face* rasterization. Leaf node indexes are emitted twice back-to-back: first for *front face*, then for *back face* rasterization.

Visiting all occupancy histogram tree nodes recursively in visibility order proceeds in the following (standard) manner. Given a view point

(for perspective projection), or a view direction (for orthogonal projection), we compute in which of eight possible *octants* it lies. This octant then determines the order in which child nodes have to be visited.

Relation to rasterization. This approach enables the subsequent rasterization stage to generate ray segments in front-to-back order, and achieve the desired property that smaller boxes *inside* larger boxes are *preceded* by the front face of the enclosing larger box, and are *succeeded* by its back face. This is illustrated in Fig. 5. This property is fundamental to achieving one of the major properties of the *Sparse-Leap* algorithm: that the occupancy class of smaller occupancy geometry bounding boxes correctly overrides that of larger bounding boxes.

7 RAY SEGMENT LIST GENERATION

Given the occupancy geometry and the computed visibility order, we can now rasterize this geometry to create the corresponding *ray segment lists*, such as the ray segment list of the ray depicted in Fig. 5. Rasterizing the occupancy geometry results in a sequence of *ray events*. These events are processed directly during rasterization (in the fragment shader), during which ray events may be merged or deleted. The remaining ray events then form the resulting ray segment list.

7.1 Ray Events

Each fragment generated during occupancy geometry rasterization results in the creation of a *ray event*. Each ray event corresponds to the intersection of a ray with one of the six faces of an occupancy geometry bounding box, at a certain depth (position) along the ray. Each box intersected by a ray produces two ray events: one for the intersection with a *front face* of the box, where the ray enters it, and one for the intersection with a *back face* of the box, where the ray exits again (Fig. 5). Each ray event therefore has an associated *event type*, which depends on whether the fragment resulted from the rasterization of a front face or a back face. Each ray event contains the following:

- *Depth.* The position (1D parameter) where the event was generated, i.e., where the ray has intersected a bounding box face.
- *Event type (entry or exit).* We store whether the event resulted from entering a bounding box or from exiting (leaving) it, respectively. Bounding box front faces produce *entry* events; back faces produce *exit* events. This distinction is crucial for the correct on-the-fly merging and deletion of events (see below).
- *Occupancy class.* The occupancy class of the event can be either *empty*, *non-empty*, or *unknown*. The occupancy class associated with an event depends on whether the event is of type *entry* or *exit*, respectively. For an entry event, the occupancy class is set to that of the bounding box that generated the event. For an exit event, it is set to the occupancy class of the box's *parent*.

Correspondence to occupancy histogram tree traversal. The idea of associating a different occupancy class with a ray event depending on whether it is an entry- or an exit event is simple, but it actually is what enables completely sequential, non-hierarchical “traversal” of the occupancy histogram tree. Without this approach, whenever an

entry event is parsed, the current occupancy class would have to be pushed onto a stack. Likewise, whenever an *exit* event is parsed, the occupancy class would have to be popped off the stack again.

This point is important to emphasize: For each ray (output pixel), traversing the corresponding linear list of ray events is *equivalent* to hierarchically traversing the occupancy histogram tree along that ray. However, in our rasterization stage, there is no tree to traverse, and there is no need for a stack, or stack-less hierarchy traversal, which would usually be required to traverse tree nodes in visibility order.

7.2 Ray Segment Lists

Each ray segment list is a singly-linked list for a given pixel that stores ray events in front-to-back visibility order. Ray events and segments, as they result from occupancy geometry, are illustrated in Fig. 5.

A ray segment is the *interval* bounded by two consecutive ray events. Corresponding to the rasterization that produces each ray event as described above, each ray segment is thus started either by a front face or by a back face of a bounding box in the occupancy geometry.

Initial ray segment list. The default ray segment list for each pixel is a single segment with occupancy class *unknown* that starts at the volume bounding box front face position and ends at the volume bounding box back face position. This list comprises the corresponding two ray events: entering the volume bounding box, and exiting it, respectively.

7.3 Occupancy Geometry Rasterization

The occupancy geometry array is rasterized in a single rendering pass. For each fragment that is generated during rasterization, we invoke a fragment shader that generates the ray segment list for that fragment. Detailed pseudo code is given in the supplementary material.

Determining the event type. The first task of this shader is to determine whether the fragment has resulted from the rasterization of the front face, or the back face, of a bounding box. This determines whether the corresponding ray event should be created with type *entry* or *exit*, respectively. This event type has to be compared against the flag stored in the occupancy geometry array, which kind of face should be retained: fragments from front faces, or fragments from back faces.

7.3.1 Ray event merging and deletion during rasterization

In order to speed up the subsequent ray-casting stage, it is crucial to avoid an unnecessarily high depth complexity (i.e., total number of events per ray) in the ray segment lists. We achieve this by performing on-the-fly merging and deletion of ray events during rasterization.

Our rules for processing ray events *at the same position* (depth) are:

- Two events of the same type (*entry* or *exit*) can always be *merged*, retaining only the second event, and overwriting the first one. The surviving occupancy class will be that of the second event.
- Two events where the first is of type *exit*, and the second of type *entry*, can *both* be *deleted* when the occupancy classes of the second event and the event *before* the first event match. This happens for consecutive segments of the same occupancy class.
- Two events where the first is of type *entry*, and the second of type *exit*, can *both* be *deleted*, independent of the corresponding occupancy classes. This happens when a ray grazes a bounding box edge or corner, generating two events at the same depth.

These rules can be executed simply for each incoming event (rasterized fragment) in the fragment shader whenever a fragment is processed, accessing the tail of the ray segment list built so far. Nevertheless, since each rule transforms a consistent ray segment list into a likewise consistent, updated ray segment list, this incremental update strategy results in the desired, consistent ray segment list for any arbitrarily long sequence of incoming events (rasterized fragments).

The second rule above will implicitly *grow* multiple consecutive segments of the same occupancy class into a single longer segment. Additionally, in order to handle boundary cases that would otherwise result in an unnecessarily high depth complexity, we detect and remove all very short *empty* ray segments that cannot be merged into a longer empty segment because they precede a *non-empty* segment.

8 RENDERING

In our implementation, volume rendering is performed via ray-casting in a ray-guided volume rendering framework [14]. Adding *SparseLeap* to the existing code was quite easy due to our conceptual separation of rendering and empty space skipping. Ray-guided volume rendering is built on the basic principle of lazy evaluation. A data brick is only loaded into memory when the brick is hit by the ray-caster. Whenever the ray-caster hits a brick that has not yet been loaded, a *data cache miss* for that brick is generated, which, in turn, triggers that the brick will be loaded into memory. In *SparseLeap*, we extend this concept further from the sole loading of volume data to the loading and computation of the meta-data required by empty space skipping.

8.1 Ray Traversal

When the ray segment lists have been created as described above, the actual ray-casting process only requires a very small modification from a standard ray-casting loop. The standard loop that iterates from sample to sample is augmented by an additional outer loop iterating over the ray segment list. This outer loop iterates linearly from ray segment to ray segment along the ray. We do this by looking at consecutive (overlapping) pairs of ray events in the ray segment list. Segments of occupancy class *empty* are simply skipped, i.e., the ray position is advanced to the subsequent segment. Segments of occupancy class *non-empty*, or of occupancy class *unknown* (they could be either empty or non-empty, without knowing which), need to be sampled. Sampling is performed exactly as in standard ray-casting, in any way desired.

8.2 Cache Misses and the Occupancy Histogram Tree

Our implementation of *SparseLeap* utilizes two different types of cache misses. The first one supports *output-sensitive* culling and empty space skipping. The second one is standard in ray-guided volume rendering, except for our use of the occupancy class *unknown*.

Occupancy misses. For segments of occupancy class *unknown*, the ray-caster generates an *occupancy miss*, so that the actual occupancy class (*empty* or *non-empty*) will be determined via culling. This leads to an output-sensitive (i.e., ray-guided) approach with *delayed* occupancy class updates: Volume bricks that are never intersected by view rays (because they are occluded, or because they are outside the view frustum) will not even get their occupancy class determined. The latter is crucial for large data, because this means that their culling meta-data need not be fetched or computed, culling need not be performed, and occupancy class changes need not be updated in the occupancy histogram tree. Finally, sub-trees in the occupancy histogram tree that have become homogeneous due to occupancy class updates, i.e., where all nodes are now of the same occupancy class, are substituted on-the-fly by a single node of that occupancy class. Determining homogeneous sub-trees is trivial using the occupancy histograms. Fig. 6 illustrates these *delayed* updates of occupancy classes in the occupancy histogram tree, driven by ray-casting and occupancy misses.

Data cache misses. Once the occupancy class of a volume brick has been determined (which happened to resolve an occupancy miss), further behavior depends on whether or not the actual volume data of the brick are resident in the volume cache (a 3D texture [14]). If the volume data are not resident *and* the occupancy class of the brick is *non-empty*, a *data cache miss* will be generated, which, as in standard

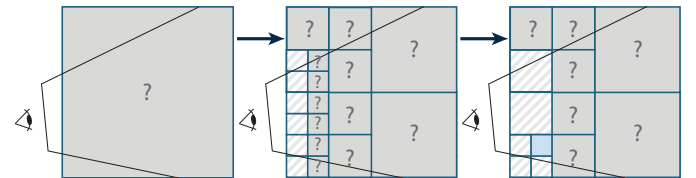


Fig. 6. **Delayed occupancy class updates.** Left: Initially, everything is *unknown* (the occupancy histogram tree comprises only the root node). Center: Some nodes are now known to be *empty*. Note the subdivisions. Right: As more nodes become known, the larger homogeneous regions now known to be *empty* can be represented by single nodes.

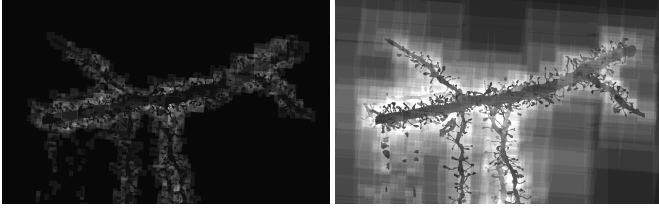


Fig. 7. **Space subdivision.** We visualize the depth complexity of empty space skipping traversal as pixel brightness (brighter means more steps). For *SparseLeap* (left) this is the number of segments ($\min = 1, \max = 31, \text{avg} = 3.84$). *SparseLeap* fragments space very little. For octree-based empty space skipping (right) this is the number of intersected nodes ($\min = 4, \max = 45, \text{avg} = 13.44$). Octrees incur a much more fragmented subdivision of space. Data set: SEM Mouse Cortex.

ray-guided rendering will lead to the data being loaded. In all other cases, no data cache miss will be generated. These are the cases when the occupancy class is *empty* or *unknown*, or the data are already in the cache. The reason for not generating a data cache miss in the *unknown* case (where data will be sampled, if in the cache, see above) is that we want to avoid burdening the system with loading new data that is not yet known to be needed. In this case, the occupancy miss will simply be resolved first, and only then will data be loaded, if actually needed.

Simplified implementation. While we have just described two different types of cache misses, the entire required functionality between GPU and CPU can be implemented by the ray-caster reporting a single, unified type of miss (essentially consisting of a brick ID). The cache miss semantics described above leave only one choice for the conceptual type of miss “meant” by the ray-caster in each case, so it is always clear whether to resolve an occupancy miss or a data miss.

8.3 Cache Usage and the Occupancy Histogram Tree

For fully output-sensitive culling and empty space skipping, we track usage information for nodes in the occupancy histogram tree, and retire unused nodes over time. In our implementation, we combine this with the usage reporting of the out-of-core memory management system [14]. The ray-caster reports which parts of the volume it still needs to be cached in the volume cache. From this, the corresponding nodes in the occupancy histogram tree are computed. Unused nodes are retired regularly using a least-recently used (LRU) scheme. This approach keeps the size of the occupancy histogram tree in correspondence with the currently visible part of the volume (the *working set* in cache terminology). In this way, the occupancy histogram tree can be made to scale with the current output size instead of the whole volume.

9 IMPLEMENTATION

Our implementation is integrated with a large-scale out-of-core ray-casting system [14]. We use OpenGL with GLSL shaders for implementing the entire *SparseLeap* pipeline. For the rasterization of the occupancy geometry into ray segment lists, fragments need to be processed in the same order as they are submitted to OpenGL, according to the occupancy visibility ordering (Sec. 6.2). In order to be able to guarantee this, the fragment shader executed during the rasterization of the occupancy geometry makes use of the *fragment shader interlock* capability of recent GPUs. This capability is exposed in OpenGL by the ARB extension `ARB_fragment_shader_interlock`, whose usual main application is order-independent transparency. For evaluation and all timings that we report, we have used an NVIDIA GeForce Titan X (Pascal) GPU with 12 GB RAM, which supports this extension.

9.1 Data Management and Culling Granularity

Our goal is to decouple the empty space skipping architecture from the actual volume rendering architecture. We therefore conceptually separate the meta-data needed for empty space skipping from the actual volume data. In our case, meta-data refers to aggregated information for box-shaped volume regions, such as the min/max value of the contained scalar data, and a set of IDs of contained segmented objects.

Meta- and Volume Data Subdivision. Separating the handling of meta-data from the actual volume data allows our implementation to use a *different granularity* for skipping empty space than that for managing volume data. Especially for finely detailed structures, empty space skipping ultimately needs to be done with a finer granularity (e.g., 16^3 blocks) than a brick size that is still feasible for memory management (e.g., 32^3 blocks). We can currently use a granularity as small as 4^3 blocks for empty space skipping of terabyte-sized volumes.

10 EVALUATION AND RESULTS

We evaluate *SparseLeap* using the volume data sets listed in Table 1, and compare against a reference implementation of empty space skipping using octree traversal, which we implemented in the same framework [14]. Our approach scales better to finely detailed, sparse data.

10.1 Data Sets

Table 1 gives information on resolution and size of the data sets that we have used. All volumes are segmented, and the table lists the number of segmented objects (segments), and the occupancy (the percentage of non-empty voxels vs. the total number of voxels) of each volume.

Connectome data set: SEM Mouse Cortex. A major motivation for developing *SparseLeap* was the mouse cortex data set depicted in Fig. 8. It is a high-resolution SEM (scanning electron microscopy) volume that our collaborators in neuroscience have manually segmented sparsely, i.e., only a few thousand select structures were segmented instead of densely segmenting millions of structures in the SEM. Neuroscientists analyze these volumes by zooming in on interesting areas, and either examining individual segments, or using interactive visual queries [2] to rapidly select different subsets of segments that are of interest, such as axons connected to the red dendrite depicted in Fig. 8.

10.2 Comparison to the State of the Art

We evaluate our novel algorithm by comparing it to two other approaches: (1) rendering without empty space skipping as a baseline, and (2) state-of-the-art image-order empty space skipping using an octree. For the latter comparisons, we have implemented octree-based empty space skipping, since this is the de-facto standard for volume rendering of large data. Usually, ray traversal (octree traversal) and empty space skipping are tightly linked: Rays intersect octree node boundaries, and at that stage determine if the block should be sampled or whether it can be skipped. The two major drawbacks are: (1) A lot of tree traversal down (and maybe up) the tree is incurred; (2) It is usually not possible to skip multiple consecutive empty nodes in one step. Each node has to be visited in turn, advancing the ray position to the next node’s boundary after the current node has been traversed (sampled or skipped). A lot of unnecessary ray-box intersections are computed. Different octree traversal approaches incur different amounts of overhead [9, 16, 17, 37]. We have implemented the octree traversal during ray-casting using the standard kd-restart algorithm [9].

Fig. 7 contrasts the qualitatively very different approaches to subdividing space for empty space skipping that our method employs vs. the fragmented subdivision incurred by a standard octree approach.

10.3 Performance

We compare overall frame rates, as well as the overall depth complexity incurred by accessing the empty space skipping data structure.

10.3.1 Rendering performance

Table 1 compares frame rates for ray-casting using the three evaluated approaches: ray-casting without empty space skipping, ray-casting with standard octree-based empty space skipping, and ray-casting using *SparseLeap*. We report frame rates for two different volume occupancy settings that allow us to compare performance between dense and sparse volume scenarios. It can be seen that our approach always outperforms ray-casting without empty space skipping. However, the most significant performance gains are obtained for sparse volumes.

Fig. 8 (left) and Fig. 9 give a detailed evaluation for a single data set (the SEM Mouse Cortex). We show the performance impact of different block sizes (Fig. 8) and data resolution levels (Fig. 9), for two

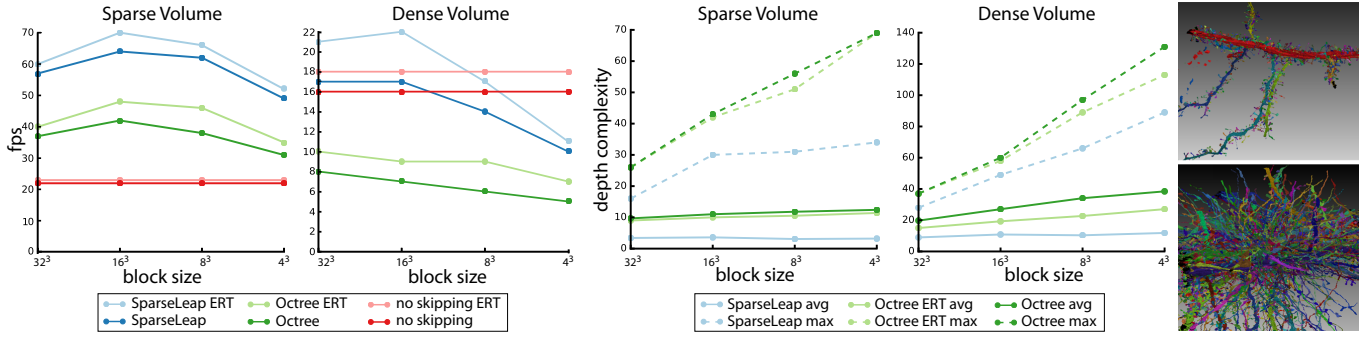


Fig. 8. **Performance comparison for different culling granularities.** Left: Frame rates for sparse (see right, top image) and dense (see right, bottom image) volume occupancy settings. We compare *SparseLeap* against an octree-based approach, with and without early ray termination (ERT), for block sizes of 4^3 , 8^3 , 16^3 , and 32^3 . Right: Depth complexity during ray-casting. The solid line represents the average depth complexity over all rays, the dashed line represents the maximum depth complexity. (Viewport: $1,200 \times 1,200$, Geforce Titan X, data set: SEM Mouse Cortex.)

different volume occupancies (dense, sparse). In the latter, the performance of ray-casting without empty space skipping is almost the same for the sparse and dense cases, respectively. However, octree skipping can be even slower than no skipping for some resolution levels in the dense case. *SparseLeap* is more efficient in both scenarios.

Disabling rasterization. Alternatively to rasterizing the occupancy geometry every frame, and then using the resulting ray segment lists in the ray-casting shader, we can also disable the rasterization stage and re-use the previous ray segment lists. This option can be used whenever the view has not been rotated or moved, e.g., when only parameters such as lighting or sampling rate have been changed. In this case, the existing ray segment lists can be re-used without performing rasterization, leading to even higher performance. The corresponding frame rates are reported in Table 1 in the second *SparseLeap* column.

10.3.2 Depth complexity (view-dependent)

Fig. 8 (right) shows a comparison of the depth complexity during ray-casting using *SparseLeap* for different block sizes, compared with standard octree empty space skipping (lower is better). Our hybrid approach significantly limits the number of nodes that need to be accessed (rasterized) and that have to be visited during ray-casting. This leads to a lower depth complexity compared with the octree approach. We report the average depth complexity over all rays of the measured frame, as well as the maximum depth complexity (dashed lines).

In *SparseLeap*, the average depth complexity stays roughly constant when the granularity becomes finer. This is due to the fact that we incur the fine granularity only where it is required by fine structures inside the volume. On the other hand, in the octree-based approach a finer granularity always leads to a higher depth complexity. This is due to the fact that smaller blocks lead to a higher spatial fragmentation, even in completely empty regions, corresponding to Fig. 7 (right).

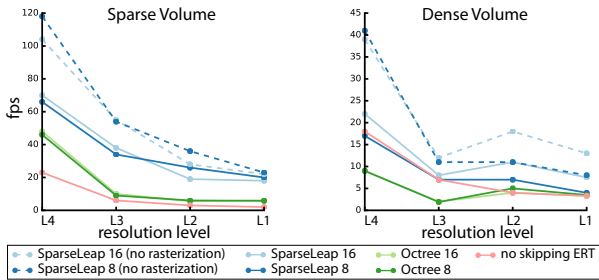


Fig. 9. **Performance comparison for different resolution levels.** We compare *SparseLeap* against an octree-based approach for block sizes of 8^3 and 16^3 , respectively, for a sparse and a dense volume (see Fig. 8, right). L4 is the lowest data resolution, L1 the highest. Dashed lines are for the performance of *SparseLeap* when the rasterization does not need to be updated, e.g., when only the light direction changes. (Viewport: $1,200 \times 1,200$, Geforce Titan X, data set: SEM Mouse Cortex.)

10.3.3 Memory requirements

The GPU memory requirements of *SparseLeap* consist of two parts. The first part is the storage required for the occupancy geometry. The second part is the storage required for the ray segment lists.

GPU geometry (occupancy geometry vs. octree). Our occupancy geometry always contains significantly fewer nodes than the equivalent octree representation, which we have directly compared using our two implementations. The reason for this is that many occupancy histogram tree nodes do not emit any geometry, whereas they do have to be stored in an equivalent octree representation on the GPU. In practice, we have observed a memory usage of just several dozen to a few hundred kilobytes for the occupancy geometry used by *SparseLeap*, whereas the equivalent octree consumed several megabytes.

GPU ray segment list buffer. The storage for ray segment lists depends on the screen resolution, and is equivalent to a frame buffer with multiple layers. The actual memory requirement depends on the average depth complexity, which, as shown in Fig. 7, is very low in *SparseLeap*. For example, an average depth complexity of 4 requires allocating a buffer of ‘four times the screen resolution’ linked list elements, plus allocating a linked list head pointer for each pixel. In our implementation, the size of each list element is 12 bytes, and the size of each list head is 16 bytes. For a screen resolution of $1,200 \times 1,200$, a buffer of 154 MB suffices up to an average depth complexity of 8.

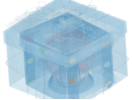
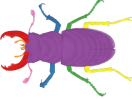
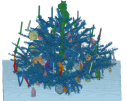
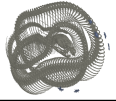
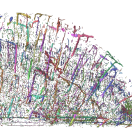
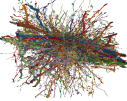
CPU memory. The CPU memory requirements between the occupancy histogram tree and a standard octree structure are very similar. Apart from standard child pointers, the histograms stored in the occupancy histogram tree consume only three integer counts per node.

11 DISCUSSION

Important characteristics that make *SparseLeap* unique: (1) Rasterizing intersections of rays with the occupancy geometry into ray segment lists moves the major cost of empty space skipping from the ray-casting stage into the faster GPU hardware rasterization. (2) Empty space skipping during ray-casting is *not* hierarchical. Each ray simply traverses the linear list of successive ray segments without costly fine-grained hierarchy traversal. (3) Temporal frame coherence (re-using occupancy geometry) as well as ray coherence (rasterization of each bounding box maps to many pixels/rays at the same time) are implicitly exploited. This amortizes the cost over successive frames as well as over nearby rays. (4) *SparseLeap* is efficient even for drastically changing sparsity characteristics due to transfer function changes and en-/disabling of segmented objects, without re-computing a space subdivision. (5) Ray segment lists conceptually *decouple* empty space skipping from volume sampling. This allows combining our approach with any out-of-core strategy and any volume representation, e.g., grids, octrees, kd-trees, AMR, page table hierarchies, and so on.

Sparsity of data. The amount and structure of sparsity in the data has a big influence on the efficiency of empty space skipping, which naturally also incurs overhead (i.e., storing and computing meta-data, look-ups in the data structure) that can only be amortized if the volume that is being rendered is sufficiently sparse. Ideally, a volume renderer

Table 1. **Data set statistics** for the volumes used for the evaluation of our method (Sec. 10). We list data resolution in voxels, storage size for original volume ('Images') and voxel-level segmentation ('Labels'), number of resolution levels (3D mipmap levels equivalent to octree levels), and number of (non-empty) segments (i.e., segmented objects) and average segment size in voxels. Occupancy in the *Segment Statistics* column refers to the percentage of all non-empty voxels (i.e., the sum of all voxels of all non-empty segments) in the whole volume. Relative to this total volume occupancy in the *Segment Statistics* column, we have measured two different *occupancy of enabled segments* settings (first row "dense," vs. second row "sparse"), by enabling different sets of segments. We compare rendering performance (frames per second) for ray-casting without empty space skipping, octree-based empty space skipping, and *SparseLeap*. For *SparseLeap*, we report two frame rates: The standard case ('w/ rast.'), and the performance when the rasterization is not re-computed ('w/o rast.'), for example when only the light source is moved. To factor out other influences such as the transfer function, all frame rates in this table have been measured without early ray termination (ERT). All timings are for a $1,200 \times 1,200$ viewport on a Geforce Titan X. All data were rendered at a resolution level and zoom factor that spanned the whole viewport.

Dataset	Description	Data Size and Type, # Resolution Levels	Segment Statistics	occupancy of enabled segments	no skipping	FPS (no ERT)		
						octree	<i>SparseLeap</i> w/ rast.	<i>SparseLeap</i> w/o rast.
	Present 492 x 492 x 442	Images: 107 MB (8 bit) Labels: 107 MB (8 bit) Resolution levels: 5	# Segments: 18 Avg size: 425.3 K Occupancy: 7.16 %	100 % 6.75 %	17 17	16 42	19 72	29 103
	Aneurysm 512 x 512 x 512	Images: 134.2 MB (8 bit) Labels: 134.2 MB (8 bit) Resolution levels: 5	# Segments: 55 Avg size: 9.25 K Occupancy: 0.38 %	100 % 4.56 %	14 14	60 112	56 120	90 183
	Stag Beetle 832 x 832 x 494	Images: 342 MB (8 bit) Labels: 342 MB (8 bit) Resolution levels: 6	# Segments: 7 Avg size: 1.93 M Occupancy: 3.96 %	100 % 12.0 %	12.5 12.5	29 47	45 83	63 129
	Xmas Tree 512 x 499 x 512	Images: 130.8 MB (8 bit) Labels: 130.8 MB (8 bit) Resolution levels: 5	# Segments: 101 Avg size: 83.94 K Occupancy: 6.49 %	100 % 3.61 %	16 16	16 47	17.5 79	26 120
	King Snake 1,024 x 1,024 x 795	Images: 833.6 MB (8 bit) Labels: 833.6 MB (8 bit) Resolution levels: 6	# Segments: 4 Avg size: 153.7 M Occupancy: 73.76 %	59.57 % 0.92 %	5.5 5.5	3 8.5	5.5 15	6.5 22
	Dreh Sensor 2,048 x 2,048 x 2,048	Images: 8.59 GB (8 bit) Labels: 8.59 GB (8 bit) Resolution levels: 7	# Segments: 85 Avg size: 9.46 M Occupancy: 9.36 %	100 % 4.65 %	6 6	13 61	10 54	13 82
	KESM Mouse Brain 2,380 x 9,216 x 2,039	Images: 44.7 GB (8 bit) Labels: 89.4 GB (16 bit) Resolution levels: 10	# Segments: 59,162 Avg size: 5.38 K Occupancy: 0.71 %	100 % 23.4 %	5 5	10 18	8 24	12 28
	SEM Mouse Cortex 21,494 x 25,790 x 1,850	Images: 1.02 TB (8 bit) Labels: 0.51 TB (16 bit) Resolution levels: 11	# Segments: 4,107 Avg size: 761.7 K Occupancy: 1.22 %	56.29 % 6.76 %	16 16	7 42	17 64	26 96

should automatically detect the point at which empty space skipping should be suspended, and switch to standard rendering for very dense volumes. We defer a detailed investigation of this issue to future work.

Moreover, for some volume structures, an optimized octree implementation might sometimes be faster (when rasterization is enabled), as can be seen in Table 1 for the Dreh Sensor data set. This, however, depends on the arrangement in space and the sizes of the enabled segments. We want to investigate this issue in more detail in the future.

Scaling to extreme-scale data. Our overall architecture is *output-sensitive* [3]. We perform culling only for potentially visible parts of the volume, which is crucial for scaling to extremely large volume data, where performing computations for large invisible parts of the volume can be too slow [14]. This was made possible via the introduction of the occupancy class *unknown*, which enables delayed culling.

12 CONCLUSIONS

Our empty space skipping approach combines object-order stages and image-order stages in a novel way to significantly reduce the traversal complexity for leaping over empty space in large volumes that con-

tain finely-detailed structures. Our method is inspired by recent order-independent transparency techniques on GPUs. However, in contrast to these methods, we sort the bounding geometry for empty space skipping on the CPU and avoid the necessity for any sorting to be done on the GPU. For this, the rasterization performed by the GPU hardware rasterization units needs to obey the ordering prescribed by the CPU traversal. This is now possible by exploiting recent GPU capabilities for enforcing the processing order of rasterized fragments. This combination of a novel algorithm together with novel GPU capabilities for order-independent transparency methods has enabled significant improvements for the volume rendering of large, complex structures.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and for pointing us to related work. We thank John Keyser for the 'KESM Mouse Brain' data [31]. 'Dreh Sensor' courtesy of Siemens Healthcare, Components and Vacuum Technology, Imaging Solutions; reconstructed by the Siemens OEM reconstruction API CERA TXR (Theoretically Exact Reconstruction). This work was supported by funding from King Abdullah University of Science and Technology (KAUST) and KAUST award OSR-2015-CCF-2533-01.

REFERENCES

- [1] R. S. Avila, L. M. Sobierajski, and A. E. Kaufman. Towards a comprehensive volume visualization system. In *IEEE Visualization*, pages 13–20, 1992.
- [2] J. Beyer, A. Al-Awami, N. Kasthuri, J. W. Lichtman, H. Pfister, and M. Hadwiger. ConnectomeExplorer: Query-Guided Visual Analysis of Large Volumetric Neuroscience Data. *IEEE Trans. on Visualization and Computer Graphics (Proc. IEEE SciVis '13)*, 19(12):2868–2877, 2013.
- [3] J. Beyer, M. Hadwiger, and H. Pfister. State-of-the-art in GPU-based large-scale volume visualization. *Computer Graphics Forum*, 8(34):13–37, 2015.
- [4] I. Boada, I. Navazo, and R. Scopigno. Multiresolution Volume Visualization with a Texture-based Octree. *The Visual Computer*, 17(3):185–197, 2001.
- [5] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Interactive 3D Graphics and Games*, pages 15–22, 2009.
- [6] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. A. K. Peters, Ltd., 2006.
- [7] T. Fogal and J. Krüger. Tuvok—An Architecture for Large Scale Volume Rendering. In *Vision, Modeling and Visualization*, pages 139–146, 2010.
- [8] T. Fogal, A. Schiewe, and J. Krüger. An Analysis of Scalable GPU-Based Ray-Guided Volume Rendering. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV '13)*, pages 43–51, 2013.
- [9] T. Foley and J. Sugerman. Kd-tree acceleration structures for a GPU raytracer. In *Graphics Hardware 2005*, pages 15–22, 2005.
- [10] E. Gobbetti and F. Marton. Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms. *ACM Transactions on Graphics*, 24(3):878–885, 2005.
- [11] E. Gobbetti, F. Marton, and J. Gutiérrez. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7):797–806, 2008.
- [12] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. In *ACM SIGGRAPH '93*, pages 231–238, 1993.
- [13] J. Gutiérrez, E. Gobbetti, and F. Marton. View-Dependent Exploration of Massive Volumetric Models on Large-Scale Light Field Displays. *The Visual Computer*, 26(6-8):1037–1047, 2010.
- [14] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE SciVis '12)*, 18(12):2285–2294, 2012.
- [15] M. Hadwiger, C. Sigg, H. Scharsach, and K. Bühler. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum (Proc. Eurographics '05)*, 24(3):303–312, 2005.
- [16] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree GPU raytracing. In *Interactive 3D Graphics and Games*, pages 167–174, 2007.
- [17] D. M. Hughes and I. S. Lim. Kd-jump: a path-preserving stackless traversal for faster isosurface raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE Visualization '09)*, 15(6):1555–1562, 2009.
- [18] R. Kähler, S. Prohaska, A. Hutanu, and H.-C. Hege. Visualization of time-dependent remote adaptive mesh refinement data. In *IEEE Visualization*, pages 175–182, 2005.
- [19] R. Kähler, M. Simon, and H.-C. Hege. Interactive volume rendering of large sparse data sets using adaptive mesh refinement hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):341–351, 2003.
- [20] B. Kainz, M. Grabner, A. Bornik, S. Hauswiesner, J. Mühl, and D. Schmalstieg. Ray Casting of Multiple Volumetric Datasets with Polyhedral Boundaries on Manycore GPUs. *ACM Transactions on Graphics*, 28(5):1–9, 2009.
- [21] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *IEEE Visualization*, pages 223–230, 2005.
- [22] A. Knoll, S. Thelen, I. Wald, C. D. Hansen, H. Hagen, and M. E. Papka. Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal. In *IEEE Pacific Visualization Symposium*, pages 3–10, 2011.
- [23] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *IEEE Visualization*, pages 287–292, 2003.
- [24] M. Labschütz, S. Bruckner, E. Gröller, M. Hadwiger, and P. Rautek. JiTTree: A just-in-time compiled sparse GPU volume data structure. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE SciVis '15)*, 22(1):1025–1034, 2016.
- [25] S. Laine and T. Karras. Efficient Sparse Voxel Octrees. In *Interactive 3D Graphics and Games*, pages 55–63, 2010.
- [26] S. Laine and T. Karras. Efficient Sparse Voxel Octrees - Analysis, Extensions, and Implementation. Technical report, NVIDIA, 2010.
- [27] S. Lakare and A. E. Kaufman. Light weight space leaping using ray coherence. In *IEEE Visualization*, pages 19–26, 2004.
- [28] E. Lamar, B. Hamann, and K. I. Joy. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *IEEE Visualization*, pages 355–362, 1999.
- [29] W. Li, K. Mueller, and A. E. Kaufman. Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering. In *IEEE Visualization*, pages 317–324, 2003.
- [30] B. Liu, G. J. Clapworthy, and F. Dong. Accelerating Volume Raycasting using Proxy Spheres. *Computer Graphics Forum*, 28(3):839–846, 2009.
- [31] D. Mayerich, J. Kwon, C. Sung, L. C. Abbott, J. Keyser, and Y. Choe. Fast macro-scale transmission imaging of microvascular networks using KESM. *Biomedical Optics Express*, 2:2888–2896, 2011.
- [32] M. Meißner, M. Doggett, J. Hirche, and U. Kanus. Efficient space leaping for ray casting architectures. In *Volume Graphics*, pages 149–161, 2001.
- [33] Z. Melek, D. Mayerich, C. Yüksel, and J. Keyser. Visualization of fibrous and thread-like data. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE Visualization '06)*, 12(5):1165–1172, 2006.
- [34] J. Mensmann, T. Ropinski, and K. Hinrichs. Accelerating Volume Raycasting using Occlusion Frustums. In *EG/IEEE Conference on Point-Based Graphics*, pages 147–154, 2008.
- [35] K. Museth. VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Transactions on Graphics*, 32(3):27:1–27:22, 2013.
- [36] V. Petrovic, J. Fallon, and F. Kuester. Visualizing whole-brain DTI tractography with GPU-based tuboids and LoD management. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE Visualization '07)*, 13(6):1488–1495, 2007.
- [37] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, 2007.
- [38] F. Reichl, M. Treib, and R. Westermann. Visualization of Big SPH Simulations via Compressed Octree Grids. In *IEEE Big Data*, pages 71–78, 2013.
- [39] D. Ruijters and A. Vilanova. Optimizing GPU Volume Rendering. In *Winter School of Computer Graphics*, pages 9–16, 2006.
- [40] L. M. Sobierajski and R. S. Avila. A hardware acceleration method for volumetric ray tracing. In *IEEE Visualization*, pages 27–34, 1995.
- [41] K. R. Subramanian and D. S. Fussell. Applying space subdivision techniques to volume rendering. In *IEEE Visualization*, pages 150–159, 1990.
- [42] M. Treib, K. Bürger, F. Reichl, C. Meneveau, A. Szalay, and R. Westermann. Turbulence visualization at the terascale on desktop PCs. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE SciVis '12)*, 18(12):2169–2177, 2012.
- [43] V. Vidal, X. Mei, and P. Decaudin. Simple empty-space removal for interactive volume rendering. *Journal of Graphics Tools*, 13(2):21–36, 2008.
- [44] M. Wan, S. Aamir, and A. E. Kaufman. Fast and reliable space leaping for interactive volume rendering. In *IEEE Visualization*, pages 195–202, 2002.
- [45] M. Wan, A. E. Kaufman, and S. Bryson. High performance presence-accelerated ray casting. In *IEEE Visualization*, pages 379–389, 1999.
- [46] G. H. Weber, M. Öhler, O. Kreylos, J. M. Shalf, E. W. Bethel, B. Hamann, and G. Scheuermann. Parallel cell projection rendering of adaptive mesh refinement data. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 8–17, 2003.
- [47] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-Of-Detail Volume Rendering via 3D Textures. In *IEEE Symposium on Volume Visualization*, pages 7–13, 2000.
- [48] R. Westermann and B. Sevenich. Accelerated volume ray-casting using texture mapping. In *IEEE Visualization*, pages 271–278, 2001.
- [49] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-time concurrent linked list construction on the GPU. *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering 2010)*, 29(4):1297–1304, 2010.
- [50] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff. Visibility Culling Using Hierarchical Occlusion Maps. In *ACM SIGGRAPH '97*, pages 77–88, 1997.