Hybrid Grids for Sparse Volume Rendering

Stefan Zellmann[*] University of Cologne
Deborah Meurer[†] University of Cologne
Ulrich Lang[‡] University of Cologne

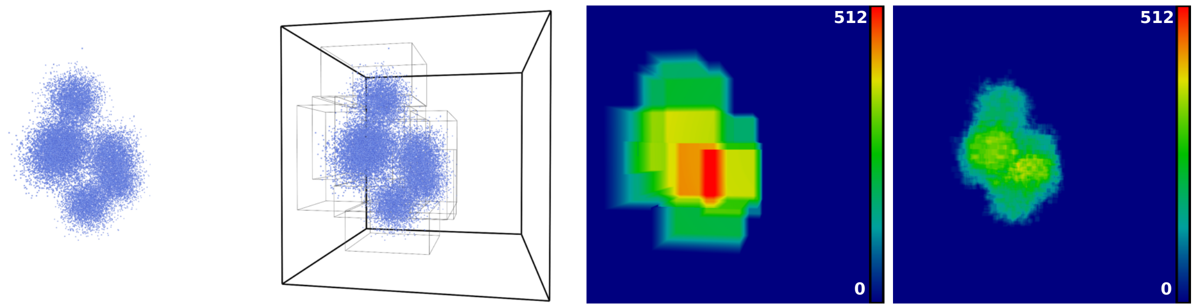


Figure 1: 3-d texture accesses performed by a ray caster using empty space skipping. The image shows a sparse 512^3 volume data set from a physics simulation (left) and the leaf node outlines of a *shallow* k-d tree (second from left). The third image shows the number of texture accesses when using the k-d tree for empty space skipping. The right image shows the number of texture accesses performed with hybrid grids, which have superior culling properties but negligible construction and storage overhead.

Abstract

Shallow k-d trees are an efficient empty space skipping data structure for sparse volume rendering and can be constructed in real-time for moderately sized data sets. Larger volume data sets however require deeper k-d trees that sufficiently cull empty space but take longer to construct. In contrast to k-d trees, uniform grids have inferior culling properties but can be constructed in real-time. We propose a hybrid data structure that employs hierarchical subdivision at the root level and a uniform grid at the leaf level to balance construction and rendering times for sparse volume rendering. We provide a thorough evaluation of this spatial index and compare it to state of the art space skipping data structures.

Index Terms: Computing methodologies—Visualization—Visualization application domains—Scientific visualization; Computing methodologies—Computer Graphics—Rendering—Ray tracing

1 Introduction

k-d trees are a popular spatial index for direct volume rendering with empty space skipping. Zellmann et al. [17] recently proposed a parallel k-d tree construction algorithm that allows to fully rebuild the spatial index in real-time for moderately sized data sets. The authors' work was motivated by the desire to interactively adapt the spatial index when the alpha transfer function changes. Their parallel construction algorithm is based on prior work by Vidal et al. [11] that generates *shallow* k-d trees with only a few (tens to hundreds) of leaf nodes. Since there are only a few of them, the non-overlapping axis-aligned bounding boxes (AABB) associated with each leaf can be efficiently sorted into visibility order up front and then traversed linearly for direct volume rendering (DVR). In the meantime, the spatial extent of typical volume data sets has however grown significantly since Vidal et al. originally published their construction algorithm. For larger data sets the culling properties of the proposed k-d tree data structure may be inferior because the *absolute* number of empty voxels contained in the few leaf nodes can become excessively high. If one adapts the parameters of the construction algorithm to allow for deeper k-d trees with significantly more nodes that better cull empty space, construction time and memory consumption increase significantly. An alternative to fully rebuild the data structure whenever the transfer function changes is to fully build up the data structure only once and then use a *min-max* range query to determine for a region if it is visible w.r.t. the current transfer function. This approach however imposes a number of restrictions and is less flexible than a data structure that just contains the desired occupancy information. Our aim is to find a data structure that has superior culling properties to shallow k-d trees and that at the same time can be built fast and is flexible. The main contributions of this paper are:

- Two alternative *hybrid* data structures combining shallow k-d trees *and* uniform grids that retain the high construction speed of shallow k-d trees and come near the rendering performance of deep k-d trees (cf. Fig. 1).
- An investigation of whether it is worthwhile to a priori build the acceleration data structure in full as opposed to rebuilding the acceleration data structure whenever the transfer function changes.
- A thorough evaluation of the various data structures. The evaluation also extends that from [17] where the authors concentrated on construction times instead of rendering performance.

The paper is structured as follows. In Section 2 we summarize related work. In Section 3 we describe a construction algorithm for our proposed *hybrid grid* data structure. In Section 4 we present performance results. Section 5 concludes this paper.

2 Related Work

Zellmann et al. [17] proposed a parallel multi-core version of the top down, greedy k-d tree construction algorithm by Vidal et al. [11]. Their trees are shallow by construction and only have a few leaves that are sorted up front and then rendered with a standard DVR algorithm. The parameters leading to this shallowness could be adjusted to yield deep trees with better culling properties but higher construction time.

[*]e-mail: zellmann@uni-koeln.de
[†]e-mail: deborah.meurer@outlook.com
[‡]e-mail: lang@uni-koeln.de

Although their culling properties are generally inferior, uniform grids [8] are still popular and are e.g. used for DVR in OSPRay [15]. They usually require an auxiliary data structure that for each possible range $[r_{min}, r_{max}]$ stores if the 1-d piecewise linear transfer function is empty in between. Wald has recently investigated data structures for such *min-max* range queries [13]. Hierarchical data structures using range queries are referred to as *min-max* trees in the literature [6, 14]. Multi-level and hierarchical grids [5, 9] are used to accelerate surface ray tracing. They are shallow and fast to construct but do not suffer from the "teapot in a stadium problem".

Zellmann et al. [16] use an *LBVH* [7] for empty space skipping and achieve real-time to interactive construction *plus* rendering performance for typical data set sizes. Hadwiger et al. [3] use rasterization to build up and merge *ray segments* from different octree levels to speed up the ensuing ray marching step. Their software system extends prior work that is focused on large scale, out-of-core volume visualization [4]. More recent work from the same research group [2] is aimed at rendering large scale microscopy data. In contrast to our primary focus, their work is not directly aimed at interactive rebuilds; the system proposed in [3] can adapt to activated and deactivated volume *segments*, but not to transfer function changes. Schneider et al. [10] use Fenwick trees that are an alternative to summed volume tables (SVT) (3-d variant of the more popular summed *area* tables) for occupancy queries that can be efficiently updated. The state of the art report by Beyer et al. [1] provides a good general overview on DVR space skipping techniques.

3 Hybrid Grids

We propose a hybrid space skipping data structure that uses the shallow k-d trees from [17] and a global uniform grid to skip over empty space at the leaf nodes. We hope that through this combination we can benefit from the fast construction time of both grids and k-d trees, while improving the traversal performance for larger volume data sets. With the original assertion by Vidal et al. that the k-d tree leaves' volume should at least be 10 % that of the volume's root node, for 1K and 2K data sets the minimal volume of the nodes would still be substantial. Relaxing the assertion would allow for smaller leaf nodes and thus for better culling properties, but would also lead to increased construction time.

3.1 Construction

The k-d tree construction algorithm consists of two phases. First a *partial* summed-volume table (SVT) is built for each 32^3 brick of the volume that conveniently fits into the L1 cache of the CPU. This phase is followed by a greedy top-down plane sweeping phase with a cost function that minimizes the sum of the two volumes of the AABBs around the non-empty voxels in each half space. With the partial SVTs, local AABBs per brick can be found in constant time. Those are then trivially combined with an $O(n)$ algorithm, where n is however not the number of voxels, but the number of bricks. For shallow k-d trees we use Vidal et al.'s original halting criterium where a leaf must have 10 % of the volume of the root node. Deep k-d trees have a minimum leaf size of 8^3 voxels. We present statistics for the various data sets that we use for the evaluation and for the two k-d tree configurations in Table 1. See Fig. 2 for an illustration of the algorithm. We propose to generate a coarse uniform grid along with the k-d tree that provides additional occupancy information. This yields two different construction schemes that we compare and that both have their individual merits.

3.1.1 Min-max grids

One implementation constructs the grid when the volume is loaded. We therefore ported the `GridAccelerator` from the open source library OSPRay [15] to CUDA. The individual grid cells only store the minimum and maximum voxel value before classification. Whenever the transfer function changes, a *range query* is used to determine if the block is empty. While OSPRay uses a naïve 2-d lookup table for the range query, we instead use an *iterative range tree* (IRT) [13] to reduce the time and storage complexity. Wald provides source code for the iterative range tree implementation as additional material that we use to extend OSPRay's `GridAccelerator`. While he reports that IRTs incur a 5 % performance degradation compared to a naïve lookup table, we found them to have no measurable performance penalty at all for transfer function arrays with 256 to 8192 entries when integrated into a DVR ray marching renderer. The parallel construction time for this data structure was however drastically reduced compared to parallel construction of a naïve table. Note that the *min-max* grid imposes a number of restrictions, namely that the transfer function is required to be one-dimensional, piecewise linear, and that voxels are required to store only scalar data.

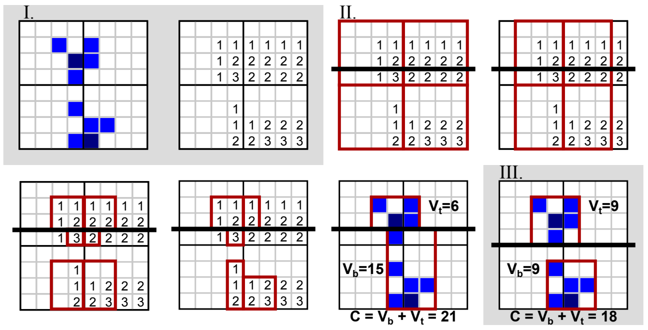


Figure 2: The k-d tree construction algorithm from Zellmann et al. [17]. I.: We compute partial summed volume tables for the four quadrants of the (2-d) volume. II.: We then use sweeping to find a plane with minimal costs. Therefore we compute local AABBs with the SVTs and trivially combine them to find their volumes. III.: Another split has better costs and will be preferred by the algorithm.

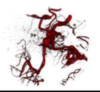
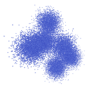
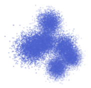
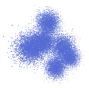
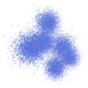
3.1.2 Pre-classified grids

A second implementation is comprised of regenerating the whole space skipping grid whenever the transfer function changes. We acquire the grid as a byproduct during k-d tree construction: the parallel construction algorithm by Zellmann et al. sets up *partial* SVTs that effectively form a uniform grid spanning the whole volume. While the partial SVTs' size of 32^3 voxels is optimized for cache utilization, we further subdivide the grid to obtain a cell size of 16^3 voxels that we found beneficial w.r.t. rendering performance. Since each partial SVT stores the occupancy information (after classification) for its cell, deriving a *pre-classified* grid that only stores the cells' binary occupancy classes is an $O(1)$ operation that can be performed on-the-fly. With this approach, we no longer need to perform a range query, which simplifies and potentially accelerates rendering and relaxes the aforementioned restrictions that the *min-max* grid imposes. On the downside, this implementation imposes extra overhead because our application implements rendering on the GPU and the grid (as opposed to the much smaller IRT) needs to be copied to GPU video memory each time the transfer function changes. This effectively results in an increased construction time. Since the additional bandwidth requirement at first glance does not seem too excessive (even for a 2K data set an extra grid of size 128^3 would be copied that only occupies a few megabyte), considering this variant may be worthwhile, since in return the aforementioned restrictions to the transfer function are relaxed.

3.2 Rendering

We also propose an optimized rendering algorithm (cf. Algorithm 1) that we implemented with NVIDIA CUDA. We first

Table 1: Statistics for the various data sets we use for the evaluation and for shallow *k*-d trees (minimal leaf volume is 10 % that of the root node's volume) and deep *k*-d trees (minimal leaf volume of 8^3 voxels).

																
Dataset	Aneurism		Bonsai		Xmas Tree		Stag Beetle		N-Body		N-Body		N-Body		N-Body	
Size	256^3		256^3		$512 \times 499 \times 512$		$832 \times 832 \times 494$		256^3		512^3		1024^3		2048^3	
Occupancy	1.01 %		6.87 %		2.90 %		4.04 %		0.14 %		0.14 %		0.15 %		0.15 %	
k-d tree	shallow	deep	shallow	deep	shallow	deep	shallow	deep	shallow	deep	shallow	deep	shallow	deep	shallow	deep
Height	12	27	8	22	11	30	7	29	8	24	8	36	8	47	10	58
# Nodes	35	3,181	25	959	33	6,873	19	8,545	21	1,895	21	8,035	21	24,303	23	38,461
Size (KB)	1	99	0.8	30	1	214	0.6	267	0.7	59	0.7	251	0.7	759	0.7	1,201

Algorithm 1 Rendering: we traverse the *k*-d tree once on the CPU to compute a sorted list of leaf nodes. We then iterate through the list and use the grid to skip over empty space inside the leaves.

```
procedure RAYCASTING(Cam, Leaves, Grid)          ▷ On the GPU
    Ray ← MAKEPRIMARYRAY(Cam)
    for each L ∈ Leaves do
        if INTERSECTS(Ray, L) then
            while Ray.ISINSIDE(L) do
                Pos ← POSITIONINGRID(Grid, Ray)
                if ISEMPTY(Grid,Pos) then
                    ADVANCETONEXTCELL(Grid, Ray)
                else
                    OutColor += INTEGRATECELL(Grid,Pos)
                end if
            end while
        end if
    end for
end procedure

procedure RENDER(HybridGrid)                     ▷ On the CPU
    SortedLeaves ← TRAVERSE(HybridGrid.KdTree)
    RAYCASTING(Cam, SortedLeaves, HybridGrid.Grid)
end procedure
```

traverse the *k*-d tree on the CPU to get the list of sorted, non-overlapping leaf node AABBs. We then discard the rest of the *k*-d tree and traverse only the leaves.

With a ray marching renderer, we can now either invoke a single CUDA kernel per AABB, or alternatively, we can transfer the list of AABBs to the GPU and let each ray intersect all the leaves and only integrate over the volumetric region of those nodes that the ray actually intersects. While the first approach implies extra communication overhead to schedule kernels from the host, the second approach may result in unnecessary ray / box intersections that could have been ruled out up front. Shallow *k*-d trees have only a few leaf nodes anyway and we expect neither overhead to be substantial. The first option is actually the one that Vidal et al. employed. GPU architectures have however changed tremendously since the authors have performed their tests, and we experimentally found *the second approach* to be superior on contemporary hardware. Regardless of how we traverse the leaves, inside the ray marching loop we use the grid to skip over empty cells by either using a range query, or by using a direct lookup if the grid is pre-classified.

Note that this approach is only viable for shallow *k*-d trees. For deep *k*-d trees with many nodes, we rather let each ray traverse the whole *k*-d tree individually and integrate the volume in a single sweep through the *k*-d tree like in [16]. For *shallow k*-d trees this approach however experimentally always turned out to be inferior to the approach outlined before.

4 RESULTS

We evaluate construction and rendering times of shallow *k*-d trees, of deep *k*-d trees with a maximum leaf size of 8^3, of uniform grids, and of the two hybrid grid data structures. The results we report in Fig. 3, Fig. 4 and Fig. 5.

4.1 Test Setup

For our tests we use the data sets depicted in Table 1: four publicly available data sets, and a data set from an *N*-body physics simulation with output resolutions of 256^3, 512^3, 1024^3, and 2048^3 voxels, respectively. We run our tests on a dual CPU system (Intel Xeon Gold 5122, eight physical cores total) and an NVIDIA Titan-V GPU with 12 GB GDDR RAM. Our iterative range tree implementation is essentially the one by Wald, but without dynamic memory allocation and with trivial OpenMP parallelization when building up the tree levels. A thorough evaluation of this data structure is obviously not within the scope of this publication; we anyway report construction times for the following array sizes to provide a rough estimate ([size/sec]): [256/0.006], [512/0.007], [1024/0.042], [2048/0.302], [4096/2.902]. Note that we keep the IRT in on-chip CUDA *shared memory*, which limits the maximum transfer function size we can use. A sensible choice depends on the distribution of high frequencies in the transfer function. We arbitrarily use an array with 1024 entries, which implies a constant overhead of 0.042 seconds for the data structures that perform a *min-max* range query. We compare naïve ray integration without space skipping, empty space skipping with the grid from OSPRay, with both shallow and deep *k*-d trees and with the two hybrid data structures. We also report rendering results with transfer functions that assign no empty space at all in Fig. 3 to assess the overhead incurred by the data structures. We further perform two synthetic benchmarks with 2K volumes to investigate how susceptible the data structures are to varying density. The first benchmark places three voxel clusters at random positions and continuously increases their size (cf. Fig. 5 top). The second one places an increasing number of fixed-size voxel clusters at random positions (cf. Fig. 5 bottom). Both benchmarks represent different spatial arrangements. With the second benchmark, when the density increases, empty space will manifest itself as holes inside the volume; this is a configuration where *k*-d trees are known to be ineffective [11, 17].

4.2 Discussion and Limitations

Our tests indicate that hybrid grids can lead to substantially improved rendering times for data set sizes above 1K, where shallow *k*-d trees fail to sufficiently cull empty space. Since the summed volume table used by the construction algorithm already induces a uniform grid, using that for space skipping incurs literally no overhead. Note that the construction times also include data copies to the GPU; we do no explicitly state those because they consistently made up for < 1 % of the overall construction times. Because of

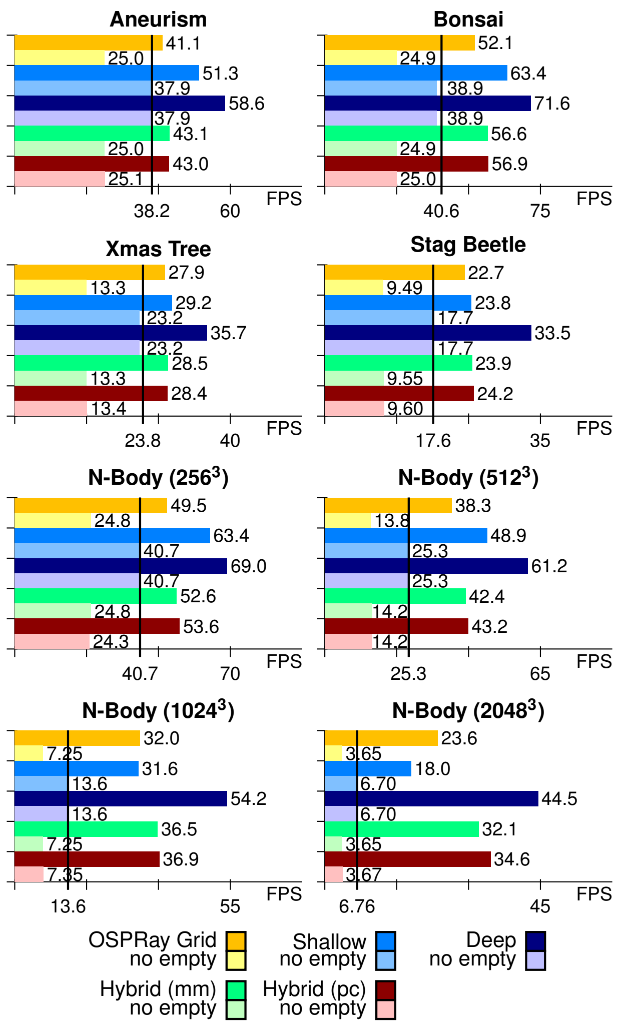


Figure 3: Rendering times in FPS (viewport: 2,160 × 2,160) for the CUDA OSPRay grid reimplementation, for shallow *k*-d trees (minimum leaf size 10 % of the volume size), deep *k*-d trees (minimum leaf size 8^3 voxels), hybrid grids with *min-max* range query (mm), and hybrid grids with pre-classification (pc). We also report results for transfer functions that assign no empty space. The black line denotes FPS achieved with naïve ray marching.

that, we come to the conclusion that pre-classified grids are always preferable to hybrid grids with *min-max* range queries. For hybrid grids to fully replace *hierarchical min-max* data structures, the construction time of the *k*-d tree however needs to be reduced for large data sets. A substantial amount of the construction time is spent at the top levels of the tree, and we believe that this overhead can be effectively reduced using binning [12]. Our benchmarks further reveal that when the density reaches a certain threshold, the *k*-d tree of the hybrid data structure consists of only a single node and the uniform grid only incurs overhead. Our synthetic benchmarks however also indicate that when empty space manifests as holes inside the volume (a worst case scenario for *k*-d trees), the hybrid data structures will effectively become a uniform grid induced over the whole volume and will have better culling properties.

5 Conclusions

We presented hybrid grids as an alternative empty space skipping data structure for DVR that combines shallow *k*-d trees with uniform grids. We proposed two grid construction schemes and found the one based on pre-classification most useful because the grid is a byproduct of the *k*-d tree construction algorithm. Hybrid grids are particularly useful for data sets that come near the size of the texture memory limit of the GPU. For smaller data sets, or if construction time is irrelevant, *k*-d trees are still preferable to hybrid grids. Uniform grids are however superior to *k*-d trees when empty space manifests as holes inside the volume. Hybrid grids are then more effective than mere *k*-d trees.

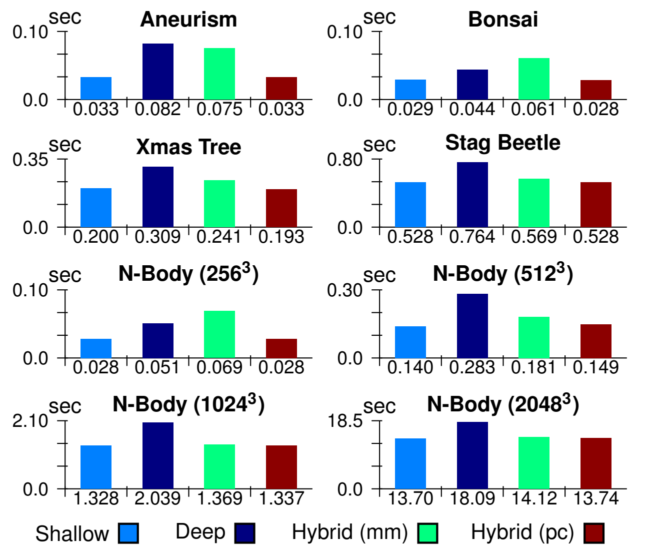


Figure 4: Construction times in seconds for shallow and deep *k*-d trees as well as hybrid grids with *min-max* range query (mm) and with pre-classification (pc).

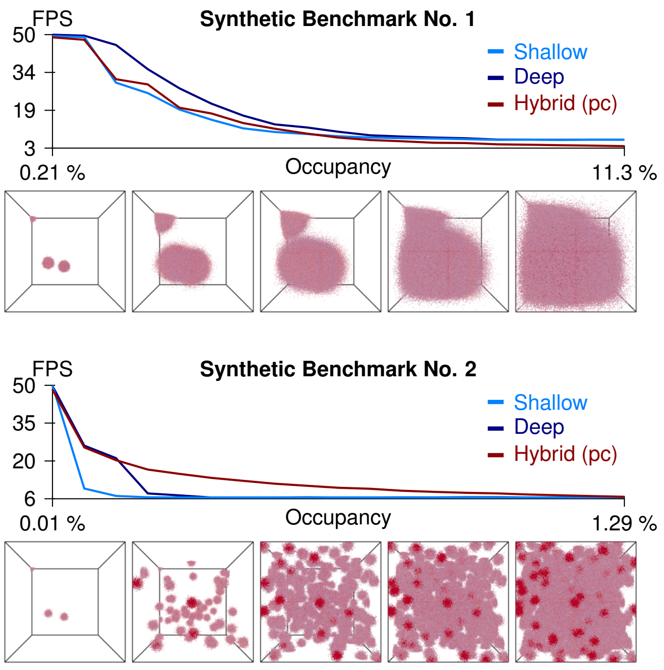


Figure 5: Synthetic benchmarks to assess which density and type of spatial arrangement cause the data structures to fail. Top: three random voxel clusters are successively increased. Bottom: random voxel clusters of fixed size are successively added to the volume.

References

[1] J. Beyer, M. Hadwiger, and H. Pfister. A Survey of GPU-Based Large-Scale Volume Visualization. In R. Borgo, R. Maciejewski, and I. Viola, eds., *EuroVis - STARs*. The Eurographics Association, 2014. doi: 10.2312/eurovisstar.20141175

[2] J. Beyer, H. Mohammed, M. Agus, A. K. Al-Awami, H. Pfister, and M. Hadwiger. Culling for extreme-scale segmentation volumes: A hybrid deterministic and probabilistic approach. *IEEE Trans. Vis. Comput. Graph.*, 25(1):1132–1141, 2019. doi: 10.1109/TVCG.2018.2864847

[3] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agos, and H. Pfister. SparseLeap: Efficient empty space skipping for large-scale volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2018.

[4] 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 Trans. Vis. Comput. Graph.*, 18(12):2285–2294, 2012.

[5] J. Kalojanov, M. Billeter, and P. Slusallek. Two-level grids for ray tracing on GPUs. *Computer Graphics Forum*, 2011. doi: 10.1111/j.1467-8659.2011.01862.x

[6] A. Knoll, S. Thelen, I. Wald, C. Hansen, H. Hagen, and M. Papka. Full-resolution interactive CPU volume rendering with coherent BVH traversal. In *Proceedings of IEEE Pacific Visualization 2011*, pp. 3–10, 2011.

[7] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 2009. doi: 10.1111/j.1467-8659.2009.01377.x

[8] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, July 1999. doi: 10.1109/2945.795215

[9] A. Pérard-Gayot, J. Kalojanov, and P. Slusallek. GPU Ray Tracing using Irregular Grids. *Computer Graphics Forum*, 2017. doi: 10.1111/cgf.13142

[10] J. Schneider and P. Rautek. A versatile and efficient GPU data structure for spatial indexing. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):911–920, Jan 2017. doi: 10.1109/TVCG.2016.2599043

[11] V. Vidal, X. Mei, and P. Decaudin. Simple empty-space removal for interactive volume rendering. *Journal of Graphics Tools*, 13(2):21–36, 2008.

[12] I. Wald. On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pp. 33–40. IEEE Computer Society, Washington, DC, USA, 2007.

[13] I. Wald. Computing minima and maxima of subarrays. In E. Haines and T. Akenine-Möller, eds., *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, pp. 61–70. Apress, Berkeley, CA, 2019. doi: 10.1007/978-1-4842-4427-2_5

[14] I. Wald, H. Friedrich, A. Knoll, and C. D. Hansen. Interactive isosurface ray tracing of time-varying tetrahedral volumes. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1727–1734, Nov 2007. doi: 10.1109/TVCG.2007.70566

[15] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navratil. OSPRay - a CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, Jan 2017.

[16] S. Zellmann, M. Hellmann, and U. Lang. A linear time BVH construction algorithm for sparse volumes. In *Proceedings of the 12th IEEE Pacific Visualization Symposium*. IEEE, 2019.

[17] S. Zellmann, J. P. Schulze, and U. Lang. Rapid k-d tree construction for sparse volume data. In H. Childs and F. Cucchietti, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2018. doi: 10.2312/pgv.20181097