

A Linear Time BVH Construction Algorithm for Sparse Volumes

Stefan Zellmann*
University of Cologne

Matthias Hellmann†
University of Cologne

Ulrich Lang‡
University of Cologne

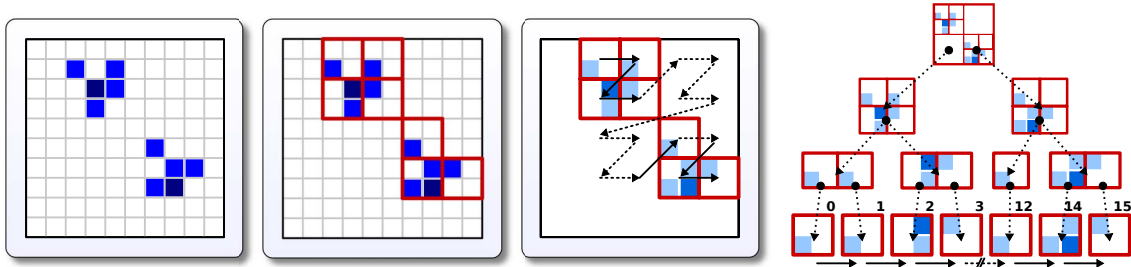


Figure 1: LBVH is a popular GPU tree construction algorithm for triangle geometry. We propose an adaptation of the algorithm for sparse volumes (left). The phases of our algorithm are comprised of first decomposing the volume into bricks and finding the non-empty ones using a parallel voting strategy (second from left). We then perform a compaction operation and sort the non-empty bricks on a z-order Morton curve (second from right). We finally build a hierarchy on the GPU by performing median splits based on the Morton codes and then propagating the leaf node bounding boxes up through the tree (right).

ABSTRACT

While fast spatial index construction for triangle meshes has gained a lot of attention from the research community in recent years, fast tree construction algorithms for volume data are still rare and usually do not focus on real-time processing. We propose a linear time bounding volume hierarchy construction algorithm based on a popular method for surface ray tracing of triangle meshes that we adapt for direct volume rendering with sparse volumes. We aim at interactive to real-time construction rates and evaluate our algorithm using a GPU implementation.

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

1 INTRODUCTION

Direct volume rendering (DVR) of sparse data defined on uniform grids can be significantly accelerated using indexing data structures like k -d trees and bounding volume hierarchies (BVH). The latter are also popular indexing data structures for triangle meshes in the research field of real-time and physically based rendering with ray tracing algorithms. The ray tracing community has long adopted the notion of *real-time rendering*, where a single frame consists of a full rebuild of the spatial index due to changes to the scene geometry *plus* rendering the frame using ray tracing. With DVR, full index rebuilds occur when the user alters the post classification transfer function. Our research focuses on fast rebuilds of the spatial index in order to reduce the overall time to render a single frame. We propose a BVH construction algorithm that is based on the LBVH algorithm by Lauterbach et al. [7] and that is suitable for real-time BVH construction for sparse volume data on GPUs. The algorithm consists of a series of $O(n)$ operations on a 3-d brick subdivision of

the uniform volume grid. The $O(n)$ operations can be implemented efficiently using parallel algorithms on the GPU.

The paper is structured as follows. In Section 2 we briefly review related research papers. In Section 3 we outline our BVH construction and traversal algorithms that we implemented for GPUs with NVIDIA CUDA. In Section 4 we provide a detailed performance analysis. In Section 5 we discuss our findings and give an outlook regarding future directions. Section 6 concludes this paper.

2 RELATED WORK

Hierarchical spatial indices for DVR can be classified into brick based [3, 8] or voxel based. The k -d tree construction algorithm by Vidal et al. [12] constructs spatial indices with per-voxel accuracy and is based on evaluating a cost function for domain decompositions found by sweeping candidate splitting planes along the three major cartesian axes and choosing the one where the cost function is minimal. This procedure is very similar to k -d tree or BVH construction for triangle geometry with the surface area heuristic (SAH) [14]. In the context of triangle geometry, it is common to build high quality trees with an $O(n \log n)$ top-down construction algorithm (where n refers to the number of triangles) and SAH. If fast rebuilds are desirable, one resorts to $O(n)$ bottom-up construction algorithms on the GPU. Fast bottom-up construction algorithms that lend themselves well to contemporary GPU architectures are the LBVH algorithm [7] and a variant thereof, the Hierarchical LBVH (HLBVH) construction algorithm [9]. We refer the reader to the following section for a more detailed overview of the original LBVH construction algorithm by Lauterbach et al. The algorithm consists of a sequence of parallel $O(n)$ algorithms which can be efficiently parallelized on a GPU where each thread is responsible for one item (e.g. a triangle or a leaf node). This sequence of algorithms is followed by a top-down hierarchy construction step that is cheap because it is based on an already existing domain decomposition. The original LBVH paper does not go into details regarding a parallel implementation of this step and the implementation presumably employed a single thread. Garanzha et al. [4] proposed a parallel version of the top-down hierarchy construction step, which can however lead to low occupancy on GPUs especially at the top-most levels of the BVH. Karras [6] proposed a fully parallel algorithm to build up the hierarchy that is based on Patricia trees.

*e-mail: zellmann@uni-koeln.de

†e-mail: hellmann@uni-koeln.de

‡e-mail: lang@uni-koeln.de

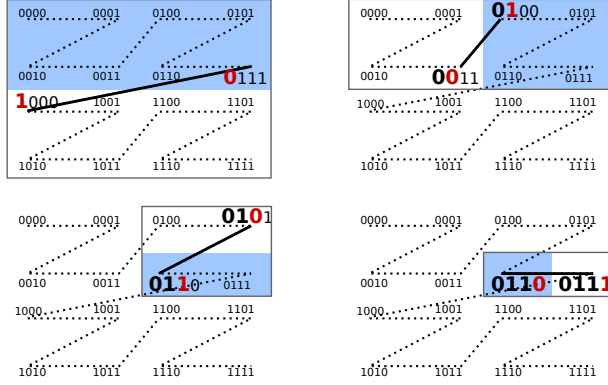


Figure 2: Median split operation using Morton codes. Split axis and position in a range can be found by searching the two Morton codes where the first most significant bits differ.

It basically consists of identifying one end of the range that an inner node overlaps, then determining if the end is the left or right one, and finally searching for the other end using binary search through a list of sorted Morton codes. The author’s algorithm compares favorably to the method by Garanzha et al. With LBVH and similar algorithms, it is possible to build spatial indices for millions of triangles within a few milliseconds on the GPU, while the resulting trees typically have inferior quality, which may result in lower rendering performance.

Fast construction and updates of spatial indices for sparse volumes have been the topic of a couple of recent research papers. The algorithm by Schneider et al. [11] uses Fenwick trees to quickly update spatial indices. The algorithm by Hadwiger et al. [5] uses a completely different approach based on rasterizing bounding boxes to skip over during volume rendering. The volume integration phase is strongly integrated with the rasterization pipeline of GPUs and ray marching of non-empty segments is performed in a shader. The authors’ approach employs a histogram tree over the whole volume that is built in advance and does not depend on the current transfer function. A second hierarchy is interactively built from the histogram tree and is used to skip over empty space. While the implementation is not focused on interactive transfer function changes, volume segments that were identified in advance can be hidden and shown interactively, and hidden segments are recognized as empty space. The algorithm by Zellmann et al. [15] is motivated by rapidly rebuilding high-quality k -d trees when the alpha transfer function changes. The authors report interactive reconstruction rates for moderately sized volumes.

3 LINEAR BOUNDING VOLUME HIERARCHIES

In this section we briefly review the original LBVH construction algorithm by Lauterbach et al. and then discuss our implementation and the necessary changes to adapt the algorithm to direct volume rendering with uniform grids. We also discuss the algorithm in terms of traversal using a ray marcher on the GPU.

3.1 Construction algorithm for triangles

The LBVH construction algorithm by Lauterbach et al. consists of a sequence of $O(n)$ algorithms in the number of input primitives. The original publication by Lauterbach et al. assumed that the primitive geometry consists of triangles. The initialization phase of the LBVH construction algorithm for triangles is comprised of first calculating the axis-aligned bounding box (AABB) for each triangle along with the AABB’s *centroid*. The centroids are then sorted on a 3-d z-order Morton curve. This procedure effectively projects the

centroids to a 3-d uniform grid. Sensible grid resolutions are e.g. 2^{10} cells for each cartesian direction, so that the linear grid index can be stored in a 32-bit integer variable. Because of the fixed bit depth of the 3-d Morton codes, sorting can be implemented using an $O(n)$ algorithm like radix sort, which can be efficiently parallelized on GPUs [10]. The LBVH algorithm exploits the Morton codes’ property that splitting positions can be found by partitioning them based on binary prefixes (cf. Figure 2). Therefore, the algorithm splits the list of sorted Morton codes into two halves at the position where the *prefix* of the binary Morton codes differs. This procedure is repeated recursively until the sequence under consideration has length one. Because the Morton codes are sorted, the splitting positions can be efficiently determined using binary search. Prefix comparisons can be performed by bitwise xor over the two indices and by means of the *count leading zeros* operation that is provided in hardware on certain architectures (e.g. on NVIDIA GPUs). After the split positions have been found, a hierarchy is constructed. This can be done using the parallel algorithm by Karras [6]. In a final step, bounding boxes for the inner nodes are constructed. This can be done in a variety of ways, with one viable option being a bottom-up phase from the leaves to the root, where each leaf’s bounding box is trivially inserted into the parent bounding box. The LBVH paper by Lauterbach et al. does not go into further details regarding the implementation of this operation.

Note that LBVH trees are effectively constructed using the *median split heuristic* (MSH), which has been shown to be inferior to e.g. the *surface area heuristic* (SAH) [13]. Also note how the triangle construction algorithm projects the primitives to leaf nodes by only considering their centroids. Depending on the scene, triangles may vary significantly in size. This may lead to significant and uncontrollable overlap between leaf nodes. Minimum overall overlap is a quality criterion for BVH trees and directly relates to rendering performance.

3.2 Construction algorithm for sparse volumes

We adapt the LBVH construction algorithm by Lauterbach et al. to generate spatial indices on the GPU for volume data defined on uniform grids. The primitive geometry assumed by the spatial index are uniformly sized volume regions (“bricks”). We discuss the necessary algorithmic changes to adapt the algorithm to volume data, and an implementation using NVIDIA’s CUDA API.

Upon initialization, when the volume is loaded for the first time, we subdivide it into bricks of size 8^3 . We copy this swizzled version of the scalar volume data to the GPU and keep it there for further use. The construction algorithm is then comprised of the following phases (cf. Figure 1):

3.2.1 Finding empty bricks

When the alpha transfer function changes, we run a parallel CUDA kernel that classifies each voxel in the grid as either being empty or not empty. All threads processing one brick then vote if the overall brick is empty by updating a common global memory location. Because GPUs lack transactional memory of any kind, concurrent writes to the same memory location result in undefined behavior, so that we implement voting through atomic write operations in shared memory for each brick. As a result we obtain a list of bricks in GPU DDR memory that are classified as either empty or not empty. Because bricks do not overlap, we can uniquely identify each one using a 3-d voxel coordinate. We therefore arbitrarily choose the minimum corner of the brick and represent it with three 32-bit integers. The list of classified bricks can thus be compactly stored using 128 bits per brick (with padding to ensure proper data alignment) that contain the coordinate and a flag indicating whether the brick is empty or not.

3.2.2 Compaction, Morton code assignment and sorting

We then perform a compaction operation to partition all the empty bricks in the list to the end so we no longer have to consider them. Then we assign 30-bit 3-d Morton codes to the non-empty bricks and sort them using a parallel $O(n)$ GPU algorithm. Note that with 30-bit Morton codes, it is possible to generate unique indices for 1024^3 bricks. For compaction and sorting we use the parallel algorithms from the C++ GPU template library Thrust [2]. We deliberately generate the Morton order *after* the non-empty bricks are determined. A separate CUDA kernel is devoted to each of the three phases.

3.2.3 Determining split positions

We use Karras' algorithm [6] to efficiently find split positions in parallel. The algorithm output consists of one list of inner tree nodes and a second list with leaf nodes (the list of leaf nodes is actually known a priori) for which parent and child relationships are set up appropriately. Some implementations optimize traversal speed by storing the two child nodes in a binary tree next to each other in memory [1]. We decided to keep the leaf nodes in a separate list for simplicity and do not implement this optimization. Shuffling the child nodes so that they are adjacent in memory would probably result in a slight performance overhead during tree construction.

3.2.4 Hierarchy generation and world space transformation

We then run a CUDA kernel where each thread is responsible for a single leaf node. The tree is traversed up to the root by following the newly determined parent pointers. The axis-aligned bounding boxes (AABBs) of the inner nodes encountered along the way are then trivially expanded to contain the AABB of the leaf node. We synchronize this operation using CUDA's `atomicMin` and `atomicMax` intrinsics. Finally, we call a CUDA kernel that transforms the AABBs to world space so that we can traverse them in a typical ray marching pipeline.

3.3 BVH traversal on the GPU

We employ a simple stack-based traversal algorithm on the GPU where each thread individually traverses a single ray through the BVH and performs volume integration over the extent of the encountered leaf nodes (cf. Algorithm 1). The algorithm is inspired by the *while-while* traversal algorithm proposed by Aila and Laine [1]. All threads start at the root and traverse the nodes they encounter along the way until they exit the volume. The inner *while* loop is executed until the thread has a leaf node to process. That leaf node is guaranteed to be the closest one not yet encountered with respect to the ray origin. Only the volume inside the leaf node is integrated over and the contribution is added to the color accumulated so far. We then offset the ray origin to the point of intersection with the far side of the leaf node's bounding box and continue traversal. By updating the ray parameter t , we ensure that the leaves are traversed in front-to-back order and that all relevant leaves were visited when the outer *while* loop finishes execution. Note that with this algorithm, the whole volume is visited within a single traversal through the tree, and we do not have to traverse through the hierarchy to obtain the next node along the ray.

When integrating the volume, we only consider absorption and emission, which keeps traversal relatively coherent because rays can be advanced in front-to-back order. It can thus be expected that the rays inside a *warp* (CUDA nomenclature for a group of threads that simultaneously execute a common set of instructions in lockstep on a GPU core) will traverse the same nodes most of the time.

4 RESULTS

We evaluate our implementation using two hardware setups, one with an NVIDIA TITAN V GPU and a second one with an NVIDIA

Algorithm 1 Stack-based BVH traversal. Each ray needs to be traversed only once through the BVH to integrate over the whole volume. The algorithm visits every node in front-to-back order. If execution exits the second while loop, we are guaranteed to process the closest leaf node.

```

procedure TRAVERSE(Ray, Root)
     $t \leftarrow 0$                                 ▷ Initialize ray parameter
    STACK.PUSH(Root)                          ▷ Initialize stack

    while not STACK.EMPTY do                    ▷ TopOfLoop
        Node  $\leftarrow$  STACK.POP
        while NODE.ISINNER do
            HITL  $\leftarrow$  INTERSECT(Ray, Node.LeftChild)
            HITR  $\leftarrow$  INTERSECT(Ray, Node.RightChild)
            if HITL and HITR then
                N  $\leftarrow$  NEAR( $t$ , Node.LeftChild, Node.RightChild)
                F  $\leftarrow$  FAR( $t$ , Node.LeftChild, Node.RightChild)
                Node  $\leftarrow$  N
                STACK.PUSH(F)
            else if HITL then
                Node  $\leftarrow$  Node.LeftChild
            else if HITR then
                Node  $\leftarrow$  Node.RightChild
            else
                goto TopOfLoop                ▷ Pop another node
            end if
        end while

        INTEGRATE(Node)                        ▷ This node is the closest leaf
         $t \leftarrow$  TFAR(Ray, Node)
    end while
end procedure

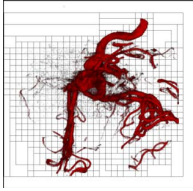
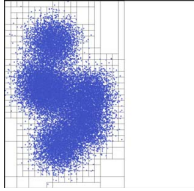
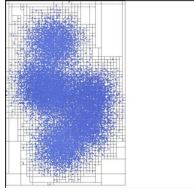
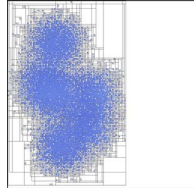
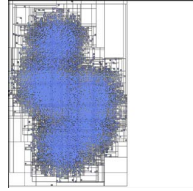
```

GeForce GTX 1080 Ti GPU. We use two data set / transfer function combinations that result in sufficiently sparse volumes: the well known PHILIPS aneurism data set, as well as a data set from an n-body simulation we computed ourselves. The aneurism data set has a resolution of 256^3 voxels, while the results from the n-body simulation were resampled on a 256^3 , 512^3 , 1024^3 , and 2048^3 uniform grid, respectively.

We are interested in the construction times of the algorithm, and also the performance during traversal. We therefore measure timing results for the different phases of the construction algorithm as well as the rendering times with the resulting trees. Our rendering setup consists of creating images with 2160×2160 pixels (the vertical resolution of a 4K display). We render with parallel projection and initially transform the camera so that the volume extent is completely visible inside the viewport. We then rotate the volume in 2° steps around the three major cartesian axes and average the individual rendering times. We also compare our algorithm to a simple ray marching implementation that does not skip over empty space. Another interesting test is the rendering performance with BVH traversal but a data set / transfer function combination that does not result in empty space at all. We expect this last test to perform worse than naive ray marching because we have to integrate over the whole volume, and the traversal overhead just adds to the integration cost.

We report the results of our performance study in Table 1 and Figures 3 and 4. The charts from Figures 3 and 4 were generated using the four differently sized n-body data sets and by averaging the performance results for the two GPU configurations. It can be seen that for moderately sized volumes, the construction times make up for only a fraction of the rendering times. With increasing volume sizes, the construction phase however becomes more dominant and eventually dominates the rendering phase for 2048^3 volumes. The

Table 1: Performance of our algorithm in milliseconds. We report LBVH construction times as well as total frame time (construction *plus* rendering with tree traversal). For comparison, we report rendering times with simple ray marching without empty-space skipping. We also report rendering times for LBVH traversal and a transfer function setup where there is no empty space to skip over.

	Aneurism, 256 ³		N-Body, 256 ³		N-Body, 512 ³		N-Body, 1024 ³		N-Body, 2048 ³	
										
	TITAN V	1080 Ti	TITAN V	1080 Ti	TITAN V	1080 Ti	TITAN V	1080 Ti	TITAN V	1080 Ti
Find Empty	0.160	0.366	0.160	0.368	1.218	2.893	8.928	18.25	82.37	190.7
Compaction	0.248	0.684	0.250	0.689	0.353	1.034	0.956	1.464	5.824	7.512
Assign Morton	0.010	0.015	0.009	0.011	0.010	0.011	0.011	0.013	0.014	0.023
Sort Bricks	0.268	1.290	0.267	1.035	0.114	0.557	0.354	0.995	0.554	1.152
Find Splits	0.149	0.540	0.144	0.428	0.159	0.406	0.316	0.792	0.418	0.924
Expand AABBs	0.157	0.099	0.155	0.089	0.278	0.226	0.586	0.604	1.804	2.208
To World	0.016	0.026	0.016	0.025	0.017	0.032	0.028	0.093	0.080	0.180
Σ Construction	1.009	3.020	1.002	2.645	2.149	5.159	11.18	22.21	91.06	202.5
Rendering	14.13	24.32	8.844	15.06	14.28	24.58	19.32	33.53	23.13	37.08
Σ Total Time	15.13	27.34	9.846	17.71	16.43	29.74	30.50	55.74	114.2	239.6
Simple Marching	42.71	51.33	19.63	33.14	38.89	67.35	80.67	143.6	157.2	299.8
No Empty Space	68.01	111.5	44.46	81.60	101.3	192.6	247.4	486.0	632.1	1300.

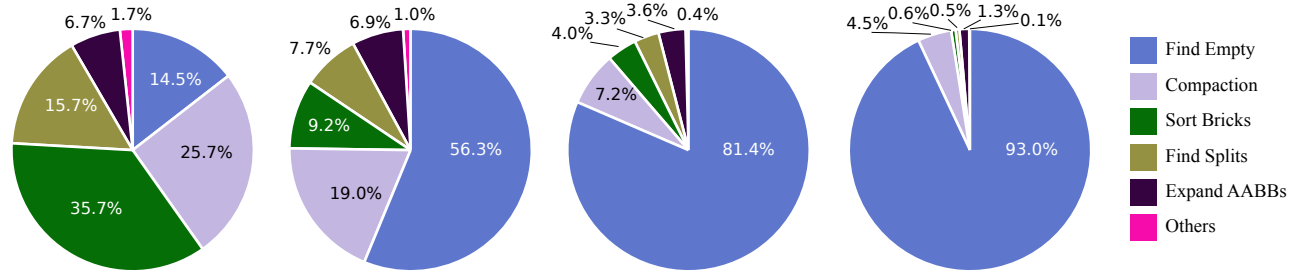


Figure 3: The most dominant phases of the construction algorithm, for (from left to right) 256³, 512³, 1024³, and 2048³ volumes. Voting for empty bricks is a per-voxel operation, while the ensuing phases operate on bricks. Note how the voting phase dominates the other phases with increasing volume size.

dominant phase during construction is the voting phase where each voxel is inspected in parallel to determine which bricks are empty. The voting phase is almost completely memory bound, since every voxel needs to be read, and then shared memory locations are updated atomically. This is reflected by the fact that voting is consistently faster by more than a factor of two on the NVIDIA TITAN V GPU that, in contrast to the NVIDIA 1080 Ti GPU, employs HBM2 high bandwidth memory. While voting is performed per voxel, the ensuing phases operate on a per-brick level and tend to become negligible for larger working set sizes. Noteworthy is the comparison of the overall algorithm, i.e. hierarchy construction plus volume integration, with simple ray marching. Even if we consider construction plus rendering a per-frame operation, the computation is still about two to three times faster than rendering with simple ray marching for 1024³ volumes. Even for sparse 2048³, where the construction time dominates the rendering time, the sum of the two is still significantly lower than just naively integrating the volume without using a BVH. Moreover, in terms of mere rendering speed, our scalability study indicates that the gain of using LBVH increases with increasing volume size. This corroborates the effectiveness of this spatial index for sparse volumes. Note however how the rendering times degrade if the spatial index is created for a data

set / transfer function combination that results in no empty space at all. This can be partially attributed to the relatively deep hierarchies and small leaf nodes of size 8³. The effect could be mitigated by pruning the spatial index so that the hierarchy is shallow and has only a few leaves. It would be interesting to see how this would affect rendering performance for sparse data sets.

5 OUTLOOK AND DISCUSSION

We have shown that the overhead for spatial index construction with the LBVH algorithm for uniform volume grids is negligible even for large data sets, and that the resulting trees are effective at removing empty space in the context of DVR. Spatial index construction for sparse volumes has traditionally been an offline process and only recent work by Zellmann et al. [15] has focused on interactive rebuilds upon transfer function updates. While their CPU algorithm is able to interactively update the transfer function, updating the transfer function *and* rendering the volume using the resulting tree does not fit into the typical frame rate budget of an interactive application. Our results suggest that with LBVH construction, both operations can be performed within 30-35 ms for currently relevant display resolutions and 1K volume grids.

In the context of surface ray tracing, one has to find a good trade-

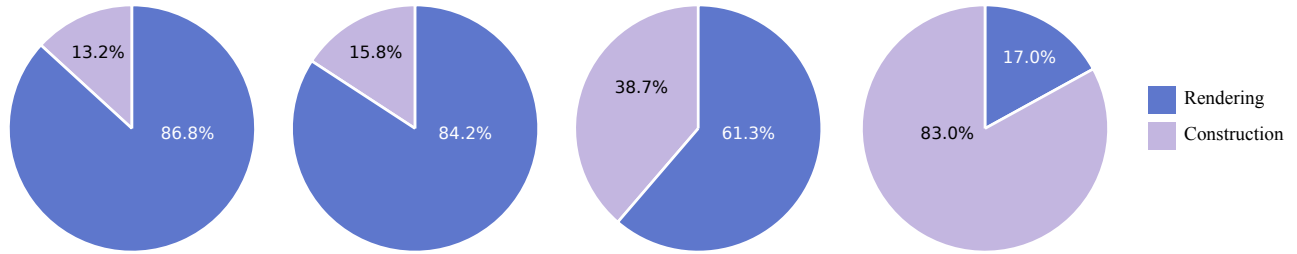


Figure 4: Relation between construction and rendering times, for (from left to right) 256^3 , 512^3 , 1024^3 , and 2048^3 volumes. With increasing volume size, the construction phase becomes more dominant.

off between spatial index construction time and the quality of the tree. LVBHs can be built within a few milliseconds for moderately sized and even large data sets, but the quality of the BVH is inferior compared to trees generated using the SAH. Top-down construction using plane sweeping and SAH will produce high quality trees, but will result in higher construction times than those achievable with $O(n)$ construction algorithms based on Morton codes.

In this sense, the tree construction algorithm by Zellmann et al. is similar to top-down BVH construction using the SAH with plane sweeping and voxel accuracy rather than brick decomposition. It would be interesting to find out how the differences in tree quality affect rendering performance. It is likely that the difference is not as severe as with triangle geometry, because nodes per construction do not overlap. The algorithm by Zellmann et al. further (deliberately) produces trees with only a few leaves. These are sorted in front-to-back order a priori, which greatly simplifies traversal because individual rays only traverse a linear list of leaf nodes. Traversal is thus effectively an “outer loop” operation, while with our LVBH algorithm, there are far too many leaves and traversal needs to be implemented as an “inner loop” operation for each ray individually. With tree pruning, it would be possible to reduce the number of leaf nodes (at the cost of bounding less empty space) to mimic the behavior of the algorithm by Zellmann et al. in this regard. We consider a comparison with this algorithm interesting future work.

6 CONCLUSIONS

We have presented and thoroughly evaluated an adaptation of the LVBH construction algorithm for sparse volumes. Our research is motivated by the necessity to be able to rapidly reconstruct a spatial index when the alpha transfer function changes. At best, the two operations *BVH construction* and *volume integration* would both fit into the frame rate budget of an interactive application, which typically requires a latency of about 30 to 35 milliseconds. We demonstrated that with our algorithm and on contemporary GPUs, this goal can be achieved even for 1024^3 volumes and currently relevant display resolutions. In a typical DVR application, the transfer function will however not change every frame, and thus tree reconstruction is only necessary once in a while. We have shown that with the spatial index based on LVBH, rendering times are consistently two to three times faster than with simple ray marching. In the future, we would like to compare our algorithm to construction algorithms that produce trees with higher quality.

REFERENCES

- [1] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pp. 145–149. ACM, New York, NY, USA, 2009. doi: 10.1145/1572769.1572792
- [2] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In W.-m. W. Hwu, ed., *GPU Computing Gems Jade Edition*, chap. 26, pp. 359–373. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.
- [3] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. ACM, ACM Press, Boston, MA, Etats-Unis, feb 2009. to appear.
- [4] K. Garanzha, J. Pantaleoni, and D. McAllister. Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pp. 59–64. ACM, New York, NY, USA, 2011. doi: 10.1145/2018323.2018333
- [5] 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.
- [6] T. Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, pp. 33–37. Eurographics Association, Goslar Germany, Germany, 2012. doi: 10.2312/EGGH/HPG12/033-037
- [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] B. Liu, G. J. Clapworthy, F. Dong, and E. C. Prakash. Octree rasterization: Accelerating high-quality out-of-core GPU volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 19(10):1732–1745, Oct 2013. doi: 10.1109/TVCG.2012.151
- [9] J. Pantaleoni and D. Luebke. HLBVH: Hierarchical LVBH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pp. 87–95. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2010.
- [10] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, pp. 1–10. IEEE Computer Society, Washington, DC, USA, 2009. doi: 10.1109/IPDPS.2009.5161005
- [11] 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
- [12] V. Vidal, X. Mei, and P. Decaudin. Simple empty-space removal for interactive volume rendering. *Journal of Graphics Tools*, 13(2):21–36, 2008.
- [13] 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.
- [14] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$. In *IEEE Symposium on Interactive Ray Tracing 2006(RT)*, vol. 00, pp. 61–69, 09 2006. doi: 10.1109/RT.2006.280216
- [15] 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