

Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction

Daniel Meister, Jiří Bittner

Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

Abstract—We propose a novel massively parallel construction algorithm for Bounding Volume Hierarchies (BVHs) based on locally-ordered agglomerative clustering. Our method builds the BVH iteratively from bottom to top by merging a batch of cluster pairs in each iteration. To efficiently find the neighboring clusters, we keep the clusters ordered along the Morton curve. This ordering allows us to identify approximate nearest neighbors very efficiently and in parallel. We implemented our algorithm in CUDA and evaluated it in the context of GPU ray tracing. For complex scenes, our method achieves up to a twofold reduction of build times while providing up to 17% faster trace times compared with the state-of-the-art methods.

Index Terms—Ray Tracing, Object Hierarchies, Three-Dimensional Graphics and Realism

1 INTRODUCTION

Ray tracing stands at the core of most image synthesis algorithms simulating light propagation. The elementary task in ray tracing is to find the nearest intersection of a given ray with the scene. To achieve high-quality results, many rays have to be traced. For example, stochastic ray tracing algorithms trace thousands of rays per pixel to reduce the noise in the synthesized image. Contemporary displays consist of millions of pixels, which in turn results in billions of rays that are tested against millions of triangles comprising the scene. Hence to solve ray tracing efficiently, we have to arrange the scene into a spatial data structure that allows accelerating ray tracing by several orders of magnitude.

One of the most common spatial data structures is the *bounding volume hierarchy* (BVH). The BVH is a tree-like structure containing scene primitives in leaves. Every node of the BVH contains a bounding volume of the geometry stored in its subtree. The most common form of the BVH for ray tracing purposes is a binary tree with axis aligned bounding boxes used as bounding volumes. There are three main approaches how to construct a BVH: incremental (by insertion), top-down (by subdivision), and bottom-up (by agglomeration). In general, the bottom-up algorithms are able to produce high-quality BVHs measured by the SAH cost [1].

Walter et al. [2] proposed the first BVH construction algorithm based on agglomerative clustering. Their method uses an auxiliary k D-tree to accelerate the nearest neighbor search. Despite the use of the k D-tree, the algorithm is not competitive with other state-of-the-art BVH construction methods regarding speed. Gu et al. [3] proposed the Approximate Agglomerative Clustering (AAC) – an efficient BVH construction algorithm using approximate agglomerative clustering that combines top-down and bottom-up approaches. This method, which uses a divide-and-conquer approach based on the Morton codes, is suitable for multi-core CPUs. Until now, it has been unclear how

to apply a similar strategy on many-core architectures such as GPU. We employ a similar idea of using the Morton codes for identifying approximate clustering. However, we use a scan-based approach combined with locally-ordered clustering to design a new GPU friendly agglomerative clustering algorithm. Our algorithm combines the idea of locally-ordered clustering with spatial sorting using the Morton codes [4]. We show that our method has low computational overhead, and it can find enough parallel work to fully utilize many cores of contemporary GPUs. As a result, the algorithm can construct a high-quality BVH faster than previous state-of-the-art methods of GPU-based BVH construction (see Figure 1). Another important feature of the method is its simplicity: the method consists of several simple steps that are executed iteratively as GPU kernels.

2 RELATED WORK

Already in the early 80s, Rubin and Whitted [6] used manually created BVHs. Weghorst et al. [7] proposed to build BVHs using the modeling hierarchy. The very first BVH construction algorithm using spatial median splits was introduced by Kay and Kajiya [8]. Goldsmith and Salmon [9] proposed the cost function known as the *surface area heuristic* (SAH). This function can be used to estimate the efficiency of a BVH during its construction, and thus it is used in most of the state-of-the-art BVH builders. The BVH construction methods require sorting and exhibit $\mathcal{O}(n \log n)$ complexity (n is the number of scene primitives). Several techniques have been proposed to reduce the constants behind the asymptotic complexity. For example, Havran et al. [10], Wald et al. [11], [12], and Ize et al. [13] used an approximate SAH cost evaluation based on the concept of binning. Hunt et al. [14] suggested to use the structure of the scene graph to speed up the BVH construction process. Doyle et al. [15] designed a hardware solution for the BVH construction based on the SAH.

High-quality BVH Great effort has also been devoted to methods which are not limited to the top-down BVH con-

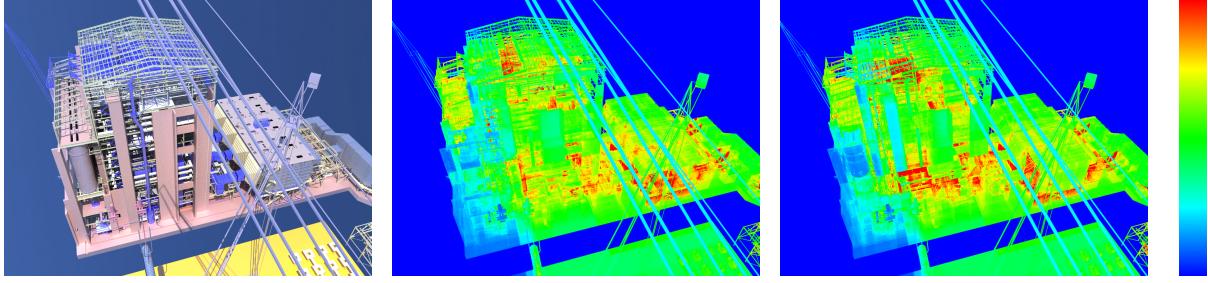


Fig. 1. GPU path tracing of the Power Plant scene (12.8M triangles) using a BVH constructed by our method (left). Visualization of the number of ray intersection operations for our method (middle) and the state-of-the-art ATRBVH method [5] (right). The red color corresponds to 325 intersections (both bounding volume and triangle intersections are counted). In this case, our method achieves 32% reduction of build time (210 ms vs. 309 ms) and 17% speedup in ray tracing performance (88 MRays/s vs. 75 MRays/s) compared with ATRBVH.

struction. These approaches allow decreasing the expected cost of a BVH below the cost achieved by the traditional top-down approach. Ng and Trifonov [16] proposed the BVH construction based on a stochastic search. Walter et al. [2] proposed to use bottom-up agglomerative clustering for constructing high-quality BVHs. Kensler [17], Bittner et al. [18], and Karras and Aila [19] proposed to optimize a BVH by performing topological modifications of an existing BVH. Aila et al. [20] identified that particularly for the methods not using the top-down approach the SAH cost metric can be corrected to correlate better with the actual trace times.

BVH modifications Dammertz et al. [21], Wald et al. [22], Ernst and Greiner [23], and Tsakok [24] proposed to use the BVH with a higher branching factor to better exploit SIMD units in modern CPUs. Ernst and Greiner [25], Popov et al. [26], Stich et al. [27], Ganestam and Doggett [28], and Fuetterling et al. [29] employed spatial splits to combine the advantages of object hierarchies and spatial subdivisions. Wachter and Keller [30], Eisemann et al. [31] devoted an effort to decrease the size of the BVH. Gu et al. [32] proposed to improve the BVH performance by adapting it to a particular ray distribution using view-dependent contraction.

Parallel BVH construction In the last decade, both multi-core CPU and many-core GPU BVH construction methods have been investigated. Wald [33] studied the possibility of fast rebuilds from scratch on the Intel architecture with many cores. Gu et al. [3] proposed parallel approximative agglomerative clustering (AAC) for accelerating the bottom-up BVH construction. Recently, Ganestam et al. [34] introduced the Bonsai method performing a two-level SAH-based BVH construction on multi-core CPUs. These two methods are considered the state-of-the-art CPU-based methods for BVH construction regarding the build time and the BVH quality.

Lauterbach et al. [35] proposed a GPU method known as LBVH based on the Morton code sorting. Pantaleoni and Luebke [36], Garanzha et al. [37] extended LBVH into the method known as HLBVH, which employs SAH for constructing the top part of the BVH. Vinkler et al. [38] proposed a GPU-based method which employs a task pool with persistent warps building a BVH in a single kernel launch. Karras [39] and Apetrei [40] further improved the LBVH algorithm; as a result, these methods are considered the fastest available GPU BVH builders. However, due to

its simplicity, the LBVH methods generally build trees of a lower quality. Karras and Aila [19] showed that a good balance between the build time and the tree quality could be achieved by a combination of LBVH and subsequent treelet optimization. This method was further improved by Domingues and Pedrini [5] in their ATRBVH method. Recently, Meister and Bittner [41] combined the k -means and agglomerative clustering in another GPU friendly BVH construction algorithm. We use the LBVH, HLBVH, and ATRBVH methods as references for the method proposed in this paper. We show that for large scenes our method improves upon the previous state-of-the-art in both the BVH build time and the corresponding trace speed.

3 BVH CONSTRUCTION VIA AGGLOMERATIVE CLUSTERING

We propose an algorithm using parallel locally-ordered clustering (PLOC) for BVH construction. The algorithm employs two main ideas: (1) We perform locally-ordered clustering on large numbers of clusters in parallel. (2) To identify suitable nearest neighbors for the clustering, we use sorting based on the Morton codes with local exploration of the neighborhood in the sorted sequence.

We first describe these two ideas in more depth and then provide the description of the complete algorithm and its implementation details.

3.1 Parallel Locally-Ordered Clustering

The agglomerative clustering algorithm starts with the scene triangles trivially forming n clusters with a single triangle per cluster (n is the number of triangles). These clusters correspond to the leaves of the BVH. Then the algorithm builds the higher levels of the BVH by merging the clusters from the lower levels.

We define a distance function d between two clusters C_1 and C_2 as the surface area A of an axis aligned bounding box tightly enclosing C_1 and C_2 [2]:

$$d(C_1, C_2) = A(\mathcal{B}(C_1 \cup C_2)) = A(\mathcal{B}(\cup(C_1, C_2))), \quad (1)$$

where $\cup(C_1, C_2)$ is the clustering operator and $\mathcal{B}(C)$ is the axis aligned bounding box tightly enclosing the geometry corresponding to cluster C . Function d obeys a non-decreasing property:

$$d(C_1, C_2) \leq d(C_1 \cup C_3, C_2) : \forall C_1, C_2, C_3. \quad (2)$$

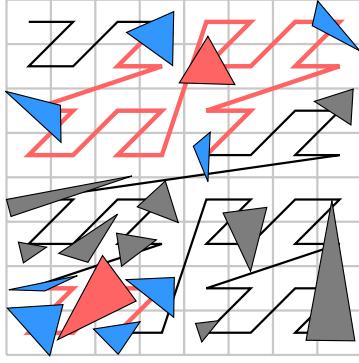


Fig. 2. Illustration of the nearest neighbor search for $r = 2$. Two clusters (red triangles) search for their nearest neighbors (blue triangles) in the neighborhood (red curve). Notice how the algorithm adapts to the density of clusters in the neighborhood.

The non-decreasing property was explored by Walter et al. [2], who proposed the *locally-ordered* agglomerative clustering algorithm. If the nearest neighbors of two clusters *mutually correspond* (these two clusters are the nearest neighbors to each other), then we know that no better neighbor will emerge in the future. Thus, we can merge the mutually corresponding clusters together. In our algorithm, we exploit this property and apply it on all pairs of mutually corresponding clusters in parallel.

3.2 Approximate Nearest Neighbor Search

The agglomerative clustering algorithm needs to identify the nearest neighbors to all current clusters. A naïve evaluation would take $\mathcal{O}(n^2)$ time for n clusters, which makes this approach very inefficient for large n . Walter et al. [2] use an auxiliary k D-tree to identify the nearest neighbors. This accelerates the algorithm, but a generalization of this approach to a parallel algorithm is difficult.

We propose a simple yet efficient way to identify the nearest neighbors for all clusters. We sort the clusters based on the Morton codes of their centroids. Then for each cluster, we use a 1D range search along the Morton curve to identify the nearest neighbors. In particular, for a cluster C_i with index i in the sorted sequence we search for its nearest neighbor in the interval $\langle i - r, i + r \rangle$, where r is the search radius. For every candidate $C_j, j \in \langle i - r, i + r \rangle \setminus \{i\}$, we evaluate the distance of the two clusters $d(C_i, C_j)$. We select the candidate C_j with the smallest distance value as the nearest neighbor. An illustration of this method for a 2D example is shown in Figure 2.

Note that this method only finds approximate nearest neighbors. However, this seems sufficient as the actual cluster pairs are found using the mutual cluster correspondence described in Section 3.1. As the Morton codes provide implicit spatial subdivision corresponding to hierarchical spatial median splits, the approximate nearest neighbor search can actually have a slightly positive influence on the BVH trace performance for some scenes. This follows from a better cluster separation for the top part of the tree that resembles top-down methods, and thus provides better correlation of the SAH cost and the trace time [20].

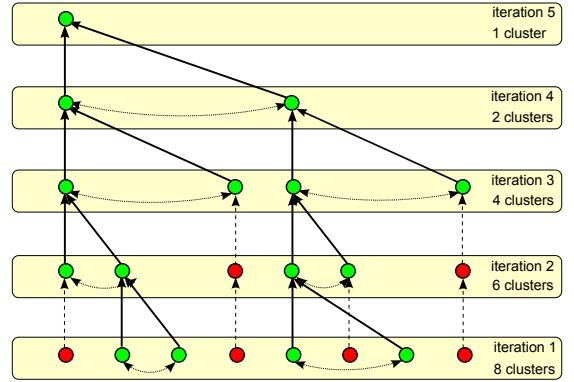


Fig. 3. Illustration of the proposed algorithm. In each iteration, we merge mutually corresponding nearest neighbors (green nodes connected by a dotted line). New clusters and not merged clusters (red nodes) enter the next iteration. The process is repeated until only one cluster remains.

3.3 Algorithm Details

The algorithm uses two buffers for storing the clusters (input and output buffer), one buffer for storing the nodes of the resulting BVH, one buffer for storing the triangle indices, and several temporary buffers.

The algorithm starts by computing the bounding boxes and the Morton codes of scene triangles. By sorting the Morton codes, we order the triangles along the Morton curve. Initially, each triangle corresponds to a single initial cluster. These initial clusters are then placed into the input buffer.

The algorithm then enters the main loop that runs through a number of iterations. In each iteration, all cluster pairs that successfully found their mutual nearest neighbor are merged. The main loop consists of three phases: nearest neighbor search, merging, and compaction. After each iteration, the input and output buffers are swapped. The main loop repeats until a single cluster remains. To guarantee that the algorithm always terminates, we prioritize the nearest neighbor with the lower index which solves a potential rare case of completely equidistant clusters (this issue will be discussed in the next section). An illustration of several iterations of the algorithm is depicted in Figure 3.

The pseudocode of the method is given in Algorithm 1. It highlights the three main phases of the algorithm:

- In the *nearest neighbor search* (shown in red), each cluster searches for its nearest neighbor in parallel using the 1D interval of clusters in the input buffer. This interval is given by the parameter r and it is clipped to prevent accessing the memory outside the buffer. Each cluster keeps the nearest neighbor found so far (based on distance function d).
- In the *merging* phase (shown in green), each cluster checks in parallel if it is equal to the nearest neighbor of its nearest neighbor. If so, both clusters are merged. To avoid conflicts, merging is performed by a thread processing the cluster with the lower index. The first cluster is replaced with the new cluster and the second cluster is marked as invalid. Simultaneously, we create an interior node corresponding to the new cluster. To determine the indices of the new

interior nodes in the node buffer, we perform a parallel prefix scan on the new clusters. We determine the actual node indices by adding the prefix scan value to the node counter.

- In the *compaction* phase (shown in magenta), we perform an exclusive parallel prefix scan to remove invalid clusters. According to the prefix scan values, we determine the positions of node indices and write them to the output buffer. It is necessary to perform a global prefix scan to remove the gaps along the Morton curve of the active clusters after some of the clusters have been merged. Note that we do not sort the new clusters according to the Morton codes of their centroids. Sorting the clusters is rather costly, and we observed that the clusters keep ordering which is sufficient for the approximate nature of our nearest neighbor search algorithm.

Algorithm 1 Pseudocode of the main loop of our massively parallel agglomerative clustering algorithm. The input of the algorithm is a sequence of n clusters C_0, C_1, \dots, C_{n-1} sorted along the Morton curve. The algorithm uses two auxiliary arrays: \mathcal{N} (for the nearest neighbor indices) and \mathcal{P} (for the prefix scan). Symbol \clubsuit denotes an invalid cluster. We assume that the prefix scan is exclusive.

```

1:  $\mathcal{C}_{in} \leftarrow \mathcal{C}_{out} \leftarrow [C_0, C_1 \dots, C_{n-1}]$ 
2:  $\mathcal{N} \leftarrow \mathcal{P} \leftarrow [0, 1 \dots, n - 1]$ 
3:  $c \leftarrow n$ 
4: while  $c > 1$  do
5:   for  $i \leftarrow 0$  to  $c - 1$  in parallel do
6:     /* NEAREST NEIGHBOR SEARCH */
7:      $d_{min} \leftarrow \infty$ 
8:     for  $j \leftarrow \max(i - r, 0)$  to  $\min(i + r, c - 1)$  do
9:       if  $i \neq j \wedge d_{min} > d(\mathcal{C}_{in}[i], \mathcal{C}_{in}[j])$  then
10:         $d_{min} \leftarrow d(\mathcal{C}_{in}[i], \mathcal{C}_{in}[j])$ 
11:         $\mathcal{N}[i] \leftarrow j$ 
12:      end if
13:    end for
14:    BARRIER()
15:    /* MERGING */
16:    if  $\mathcal{N}[\mathcal{N}[i]] = i \wedge i < \mathcal{N}[i]$  then
17:       $\mathcal{C}_{in}[i] \leftarrow \text{CLUSTER}(\mathcal{C}_{in}[i], \mathcal{C}_{in}[\mathcal{N}[i]])$ 
18:       $\mathcal{C}_{in}[\mathcal{N}[i]] \leftarrow \clubsuit$ 
19:    end if
20:    BARRIER()
21:    /* COMPACTION */
22:     $\mathcal{P}[i] \leftarrow \text{PREFIXSCAN}(\mathcal{C}_{in}[i] \neq \clubsuit)$ 
23:    if  $\mathcal{C}_{in}[i] \neq \clubsuit$  then
24:       $\mathcal{C}_{out}[\mathcal{P}[i]] \leftarrow \mathcal{C}_{in}[i]$ 
25:    end if
26:    BARRIER()
27:  end for
28:   $c \leftarrow \mathcal{P}[c - 1]$ 
29:  if  $\mathcal{C}_{in}[c - 1] \neq \clubsuit$  then
30:     $c \leftarrow c + 1$ 
31:  end if
32:  SWAP( $\mathcal{C}_{in}, \mathcal{C}_{out}$ )
33: end while

```

3.4 Algorithm Correctness

The proposed algorithm combines the approximate neighbor search with locally-ordered clustering. This poses a question on the finiteness of the algorithm, i.e. will the algorithm always find at least two clusters to be merged in the given iteration?

We show that at least two clusters are merged in each iteration: We represent the relation of the nearest neighbors as a directed graph $G = (V, E)$, where V is a set of vertices corresponding to n clusters and E is a set of n edges corresponding to nearest neighbor relation. Our goal is to show that the graph G always contains a directed cycle of length two, i.e. two vertices are pointing to each other.

We obtain an undirected graph G' from the graph G by dropping orientation of the edges E . The graph G' has to contain a cycle C' since $|E| \geq |V|$. Cycle C' must also be a directed cycle C in G because each vertex has exactly one outgoing edge. Trivially, C cannot contain cycles of length 1 (by definition of the nearest neighbor search). It remains to show that the length of C cannot be greater than 2. Suppose that the graph G contains a cycle $v_1, v_2, \dots, v_k, v_1$ for $k > 2$. The procedure of searching nearest neighbors implies:

$$d(v_1, v_2) \leq d(v_2, v_3) \leq \dots \leq d(v_k, v_1) \leq d(v_1, v_2). \quad (3)$$

This is true only if all distances are the same. In this case, we force the cycle to be of length two by preferring the neighbor with the lowest index as mentioned in Section 3.3. Thus, the graph G has to contain at least one cycle C of length two. Therefore in each iteration, at least one pair of clusters is merged.

3.5 Complexity Analysis

It is difficult to estimate the expected running time of Algorithm 1 for general input data. However, we can estimate the best and worst case running times of our algorithm. Let n denote the number of input clusters, r the search radius, and p the number of processors working in parallel. Below, we analyze the best and the worst cases for sequential and parallel versions.

In the worst case, we merge only one pair of clusters in each iteration. We perform $n - 1$ iterations; in i -th iteration, we execute $n - i + 1$ nearest neighbor search queries. Each query takes linear time with respect to r . The prefix scan takes $\mathcal{O}(n - i)$ time in i -th iteration. The worst case sequential complexity is thus $\mathcal{O}(rn^2)$. The parallel prefix scan takes $\mathcal{O}(\frac{n-i}{p} + \log p)$ time in i -th iteration. The parallel complexity is then $\mathcal{O}(r\frac{n^2}{p} + n \log p)$ assuming $p \leq n$.

In the best case, we assume that n is a power of two and all clusters find their neighbors in each iteration. Thus, we perform $\log_2 n$ iterations and i -th iteration performs $\frac{n}{2^{i-1}}$ nearest neighbor search queries. The prefix scan takes $\Omega(\frac{n}{2^i})$ time in i -th iteration. Thus, in the best case the sequential complexity is $\Omega(rn)$. In the parallel case, we need $\Omega(\frac{n}{2^i p} + \log p)$ time for parallel prefix scan in i -th iteration. The parallel complexity is then $\Omega(r(\frac{n}{p} + \log n) + \log n \log p)$. The first occurrence of $\log n$ term in the complexity follows from the necessity of performing $\log_2 n$ even for large number of processors. The $\log n \log p$ is due to the lower bound on performing $\log_2 n$ parallel prefix scans.

The experimental results indicate that the actual running times exhibit behavior close to the best case complexity bounds, i.e. linear dependence on the search radius r and the number of input clusters n .

3.6 Implementation Details

We implemented the algorithm in CUDA [42]. We use 30-bit Morton codes computed using the expanded bounding box of the scene. The expanded bounding box is computed by taking the largest extent of the scene bounding box and creating circumscribed cube around the scene. To sort the triangles along the Morton curve, we use the radix sorting algorithm proposed by Merrill and Grimshaw [43]. The main loop consists of five kernels. The first kernel corresponds to the nearest neighbor search phase. The second kernel corresponds to the merging phase. The last three kernels correspond to the compaction phase (prefix scan). In the nearest neighbor search phase, we use a shared memory cache to minimize the number of transfers of cluster bounding boxes from the global memory. This cache consists of $B + 2r$ bounding boxes, where B is the number of block threads and r is the radius mentioned above. At the beginning, threads in the block fill the cache. Then we use this cache to search for the nearest neighbors. The bank conflicts should be avoided because the size of the bounding boxes is 24 B and memory accesses are coalesced. In the merging phase, we perform a warp-wide prefix scan on the number of new clusters using the `_ballot` function. We determine the node indices for a new cluster by atomically adding the number of new clusters within a warp to the node counter. This is done by adding the values of the warp-wide prefix scan and the original value of the node counter returned by the atomic addition.

Collapsing Subtrees The resulting BVH contains exactly one triangle per leaf. Collapsing some subtrees to leaf nodes may decrease the total SAH cost [18], [20]. A GPU implementation of subtree collapsing is not trivial and therefore we briefly describe our implementation of this method. We use several passes of the parallel bottom-up traversal proposed by Karras [39]. The procedure was originally used to refit bounding boxes. We suppose that each node has a parent index, and each internal node has a counter (initially set to 0). Threads proceed up from the leaves using parent indices. In each interior node, a thread atomically increments the corresponding counter. If the original value of the counter was 0 then the thread is killed. Otherwise, it proceeds to the parent. In other words, the first thread is killed and second continues up to the root. It is guaranteed that each node is processed by a single thread, and both subtrees are already processed.

In the first pass, we perform a bottom-up traversal that marks each node as leaf or interior depending on the associated BVH cost. We compare the SAH cost of the node as a subtree and the SAH cost of the node being a leaf. If collapsing pays off, we mark the node as a leaf, otherwise as an interior node. In the second pass, we have to determine the roots of the collapsed subtrees. We perform a bottom-up traversal and track the highest leaf found so far. In the third pass, we mark all nodes in the collapsed subtree as invalid. Again we perform a bottom-up traversal until we reach the

node identified in the previous pass and mark all visited nodes as invalid. In the fourth pass, we perform a prefix scan on valid nodes to determine the new node indices. We remap the child and parent indices using the values of the prefix scan.

We determine the leaf sizes by atomically incrementing a counter associated with the leaf. Each leaf knows the offset within its segment of triangle indices as the atomic operation returns the original value of the counter. Then we perform a prefix scan on the leaf sizes to determine the bounds of continuous segments of triangle indices. Finally, we write triangle indices to the appropriate continuous segments in the triangle index buffer. The implementation of our BVH builder can be downloaded from the PLOC project site¹.

4 RESULTS AND DISCUSSION

We have evaluated the PLOC method using nine test scenes of different complexity. As reference methods, we used the LBVH builder proposed by Karras [39], the HLBVH builder proposed by Garanzha et al. [37], the ATRBVH builder proposed by Domingues [5], and the AAC builder proposed by Gu et al. [3]. For LBVH as well as HLBVH, we used 30-bit Morton codes, HLBVH used 15 bits for the SAH-based top-tree construction. For ATRBVH, we used the publicly available implementation of treelet restructuring using treelets of size nine with two iterations. For AAC, we used the publicly available sequential implementation (the comparison with AAC is presented in a dedicated section below).

For our method, we use three settings with different radius r : $\text{PLOC}_{r=10}$, $\text{PLOC}_{r=25}$, and $\text{PLOC}_{r=100}$. In all cases, we use an adaptive leaf size based on collapsing subtrees discussed in Section 3.6. We evaluated the constructed BVH using a high-performance ray tracing kernel of Aila et al. [44]. All measurements were performed on a PC with Intel Core I7-3770 3.4 GHz (4 physical cores), 16 GB RAM, GTX TITAN X GPU with 12 GB RAM (Maxwell architecture, CUDA 7.5), Windows 7 OS. For all methods, we used customized version of radix sort from CUB 1.1.1 to sort the Morton codes.

The results are summarized in Table 1. For each method, we report the SAH cost of the constructed BVH (using traversal and intersection constants $c_T = 3$ and $c_I = 2$), the average trace speed, the build time, and the time-to-image (total time) for two different application scenarios. The first time-to-image measurement corresponds to 8 samples per pixel; the second measurement corresponds to 128 samples per pixel, both using 1024×768 image resolution. Our path tracing implementation uses the next event estimation with two light source samples per hit and the Russian roulette for path termination. The reported times are an average of three different representative camera views to reduce the influence of view dependency. For the build time, we report the real execution time measured on the CPU side as well as the sum of kernel times (in brackets). The build time also includes a conversion to the data representation needed by the ray tracing kernel.

1. <http://dcgi.felk.cvut.cz/projects/ploc>

BVH quality From the results, we can see that our PLOC and ATRBVH are quite competitive. PLOC has lower SAH cost and higher trace speed for six scenes compared with ATRBVH. PLOC has lower trace speed than ATRBVH for Happy Buddha (−5%), Soda Hall (−3%), and Hairball (−16%). However, PLOC performs better for large complex architectural scenes. PLOC has higher trace speed than ATRBVH for Conference (+5%), Manuscript (+5%), Pompeii (+13%), San Miguel (+14%), Vienna (+14%), and for Power Plant (+17%). We can observe that the SAH costs stabilize quite fast even for small radii (see Figure 4).

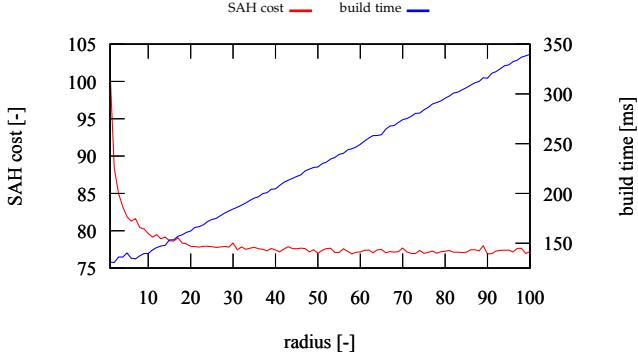


Fig. 4. The dependence of the SAH cost and build on the radius r for the Power Plant scene. Note that the build time exhibits linear dependence on r .

Build time LBVH is the fastest builder overall. HLBVH is the second fastest method for seven scenes. PLOC $_{r=10}$ and PLOC $_{r=25}$ are faster than ATRBVH for all scenes except Conference and Happy Buddha. Compared with ATRBVH, PLOC $_{r=10}$ achieves the following speedups: Soda Hall (+31%), Hairball (+24%), Manuscript (+53%), Pompeii (+47%), San Miguel (+42%), Vienna (+55%), and Power Plant (+46%). Kernel times of different phases of the algorithm are shown in Figure 5.

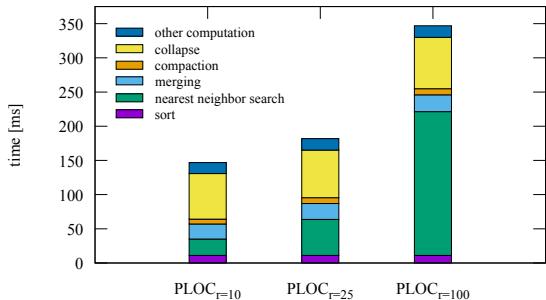


Fig. 5. Kernel times of different phases of the BVH construction for the Power Plant scene and three different radii.

Time-to-image For the low quality rendering, PLOC has lower times for six scenes compared with ATRBVH. We observe speedups for Soda Hall (+12%), Manuscript (+34%), Pompeii (+29%), San Miguel (+22%), Vienna (+39%), and Power Plant (+24%). PLOC is slightly slower than ATRBVH for Conference (−1%), Happy Buddha (−7%), and Hairball (−8%). For the high-quality rendering, PLOC is faster than ATRBVH for six scenes. We observe speedups for Conference (+4%), Manuscript (+9%), Pompeii (+13%), San Miguel (+11%), Vienna (+14%), and Power Plant (+15%).

PLOC is slower for the object-like scenes, namely Happy Buddha (−6%), Soda Hall (−2%) and Hairball (−19%).

Iterations We measured the number of iterations for various radii (see Figure 6). We observed that the number of iterations is approximately 2–3 times higher than the depth of the BVH. For almost all scenes the number of iterations slowly grows with increasing radius, and it almost stabilizes for $r > 20$. An exception is the Power Plant scene. There is a significant step down at $r = 6$. We expect this is caused by large variance in triangle sizes and subsequent need for larger neighborhood for the nearest neighbor search.

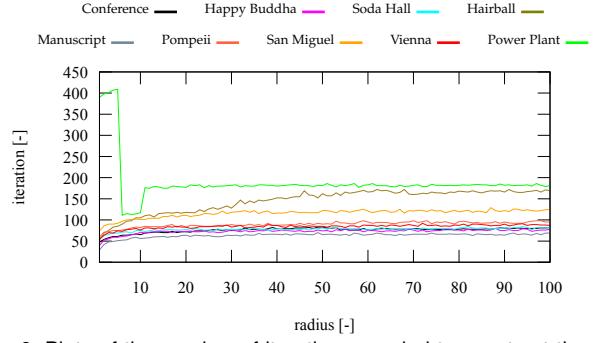


Fig. 6. Plots of the number of iterations needed to construct the whole BVH depending on the setting of the radius r .

Algorithm visualization To provide better insight into the behavior of the algorithm, we visualized the neighboring triangles determined by the 1D interval along the Morton curve (see Figure 7). We also visualized the active clusters for different phases of the BVH construction (see Figure 8).

Comparison with AAC We have conducted a comparison of our method with the state-of-the-art CPU builder – the AAC method proposed by Gu et al. [3]. Both algorithms use the Morton codes for performing approximate nearest neighbor search combined with the clustering phase. Therefore we can expect that the BVH they produce will be similar. Note, however, that the algorithms use very different computation strategies designed to follow the capabilities of different architectures, multi-core CPU vs. many-core GPU. AAC uses top-down partitioning phase while keeping relatively large computation state on the stack (including distance matrices). PLOC uses iterative parallel bottom-up locally-ordered clustering while keeping minimal state information to support efficient GPU execution.

For the comparison, we used the publicly available implementation of AAC provided by Gu et al. [3]. This

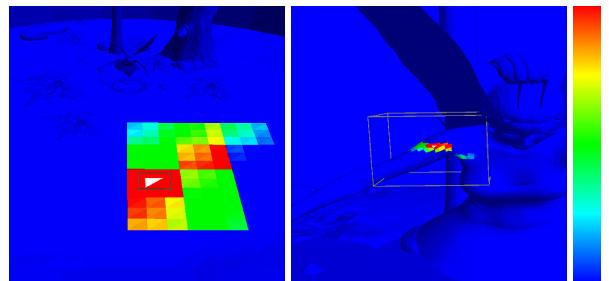


Fig. 7. Visualization of the neighboring triangles. The heat value corresponds to the distance from the white triangle along the Morton curve. Blue triangles are beyond the radius ($r = 100$).

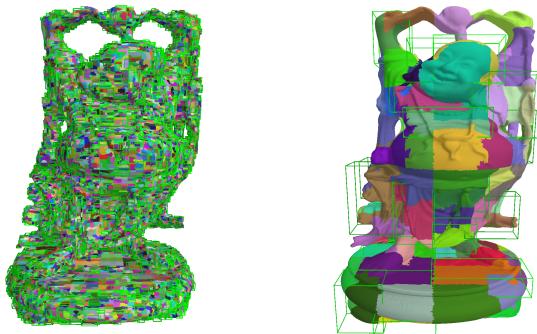


Fig. 8. Visualization of the active clusters for the Happy Buddha scene in iteration 20 (left) and 55 (right). In total, 70 iterations were needed to construct the whole BVH. The bounding boxes highlight those clusters which were formed during the depicted iteration.

implementation is sequential, and therefore we divided the running times by the number of physical cores in the testing PC (4 physical cores) to get optimistic bound on the AAC running times on this architecture. The comparison is summarized in Table 2.

Regarding the cost of the BVH, we can observe that the results are indeed very similar. An exception is the Power Plant scene for which the AAC implementation constructs a tree with significantly higher cost, possibly due to a bug in the implementation related to the size of the scene. The build times for PLOC are significantly lower than for AAC, particularly for larger scenes (e.g. 3x for $PLOC_{r=25}$ vs. AAC-Fast for San-Miguel). This can also be observed from Figure 9 that shows the dependence of build time on the number of triangles. PLOC seems to provide slightly better scalability towards very large scenes. The results indicate that for application involving GPU ray tracing our method would be the method of choice whereas for CPU ray tracing AAC is still a good option, particularly when using a powerful CPU with many cores.

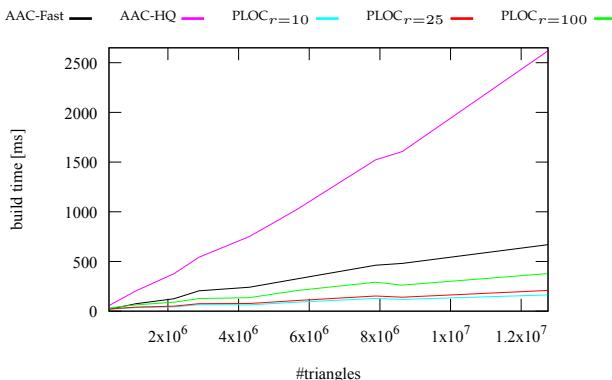


Fig. 9. Dependence of build time on the scene size (number of triangles) for the method by Gu et al. [3] (AAC-Fast, AAC-HQ) and our method ($PLOC_{r=10}$, $PLOC_{r=25}$, $PLOC_{r=100}$).

Limitations The proposed method achieves superior results for large scenes. For smaller scenes, though, it is less efficient than the tested reference methods regarding build time (e.g. for the Conference scene with 331k triangles); this mainly follows from the larger kernel management overhead. We launch several kernels for each iteration of the method, which leads to a larger number of kernels that need

to be executed compared with the reference methods. While for large scenes this overhead becomes almost negligible due to the longer kernel execution times, it remains an issue for smaller scenes. This behavior may improve in the future as the hardware vendors aim at further reduction of kernel management overhead. For now, our method provides best results for scenes larger than roughly million triangles.

Note that the available implementations of LBVH, HLBVH, and ATRBVH do not represent all state-of-the-art GPU builders. It would be interesting to compare our method also with the original implementation of the TRBVH method [19], which achieves very good build times and BVH quality. However, a direct comparison is problematic as the original implementation of TRBVH is not publicly available.

5 CONCLUSION AND FUTURE WORK

We proposed a new GPU-oriented BVH construction algorithm using agglomerative clustering based on the Morton curve ordering. The algorithm uses fast scan-based approximate nearest neighbor search combined with locally-ordered clustering. We implemented our algorithm in CUDA and compared it with the LBVH, HLBVH, ATRBVH, and AAC methods. The results show that our algorithm competes favorably with the state-of-the-art methods. In the worst cases, our algorithm is very close to the ATRBVH method; in the best cases, our algorithm achieves speedups up to 39% (time-to-image). This indicates that the proposed method is probably the fastest available BVH builder for constructing high-quality BVHs. Our algorithm is simple yet scalable and efficient. Setting a single parameter, i.e. the radius, we can easily trade the BVH construction speed for the rendering performance.

In the future, we would like to conduct a deeper study of the influence of the radius parameter r on the build times and the trace speed. Varying this parameter across the scene and also across different iterations might provide the optimal balance between the construction time and the trace time for a particular rendering scenario. The PLOC method uses standard parallel constructs, and thus it would be interesting to modify it also for the context of many-core CPUs.

6 ACKNOWLEDGEMENTS

We would like to thank our colleague Jakub Hendrich for proposing the proof of the algorithm correctness and providing other valuable comments. This research was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS16/237/OHK3/3T/13.

REFERENCES

- [1] J. D. MacDonald and K. S. Booth, "Heuristics for Ray Tracing Using Space Subdivision," *Visual Computer*, vol. 6, no. 6, pp. 153–65, 1990.
- [2] B. Walter, K. Bala, M. Kulkarni, and K. Pingali, "Fast Agglomerative Clustering for Rendering," in *IEEE Symposium on Interactive Ray Tracing*, 2008, pp. 81–86.
- [3] Y. Gu, Y. He, K. Fatahalian, and G. Blelloch, "Efficient BVH Construction via Approximate Agglomerative Clustering," in *Proceedings of High-Performance Graphics*, 2013, pp. 81–88.

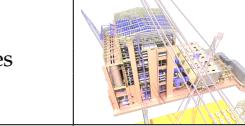
	Conference #triangles 331k						Happy Buddha #triangles 1087k						Soda Hall #triangles 2169k					
	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time ¹ [ms]	total time ² [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time ¹ [ms]	total time ² [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time ¹ [ms]	total time ² [ms]			
LBVH	154	210	3 (3)	248	3932	204	128	11 (10)	109	1573	252	220	18 (17)	115	1572			
HLBVH	118	257	11 (6)	211	3222	184	134	22 (16)	116	1518	219	252	33 (26)	118	1389			
ATRBVH	87	287	10 (9)	190	2888	169	144	34 (33)	121	1425	173	260	62 (60)	145	1376			
PLOC _{r=10}	85	298	19 (6)	192	2786	175	135	36 (21)	129	1517	179	251	43 (28)	128	1408			
PLOC _{r=25}	84	299	22 (8)	195	2784	175	137	39 (25)	131	1507	176	243	50 (35)	138	1457			
PLOC _{r=100}	85	300	31 (15)	202	2778	179	133	63 (47)	157	1575	177	242	88 (71)	177	1503			
	 Hairball #triangles 2880k						 Manuscript #triangles 4305k						 Pompeii #triangles 5632k					
	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time ¹ [ms]	total time ² [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time ¹ [ms]	total time ² [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time ¹ [ms]	total time ² [ms]			
LBVH	1233	55	23 (22)	297	4426	182	150	40 (39)	153	1843	428	81	47 (46)	293	3980			
HLBVH	1225	56	43 (35)	314	4383	134	164	64 (55)	166	1689	314	99	76 (65)	277	3289			
ATRBVH	1073	64	83 (82)	319	3869	106	194	132 (130)	217	1497	234	119	168 (167)	333	2804			
PLOC _{r=10}	1092	54	63 (43)	344	4606	91	203	62 (51)	143	1363	175	132	89 (72)	237	2462			
PLOC _{r=25}	1085	48	76 (51)	405	5285	96	199	77 (64)	160	1409	172	134	105 (88)	252	2452			
PLOC _{r=100}	1081	50	124 (92)	426	4961	101	192	134 (120)	220	1509	170	135	203 (184)	348	2516			
	 San Miguel #triangles 7880k						 Vienna #triangles 8637k						 Power Plant #triangles 12759k					
	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time ¹ [ms]	total time ² [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time ¹ [ms]	total time ² [ms]	SAH cost [-]	trace speed [MRays/s]	build time [ms]	total time ¹ [ms]	total time ² [ms]			
LBVH	252	61	64 (63)	561	8017	302	92	79 (78)	317	3902	127	55	93 (92)	746	10551			
HLBVH	184	87	100 (86)	455	5733	217	105	120 (105)	330	3483	114	65	136 (119)	685	8958			
ATRBVH	146	97	221 (219)	537	5269	147	164	262 (259)	396	2402	81	75	304 (300)	775	7909			
PLOC _{r=10}	144	105	128 (108)	420	4784	111	179	119 (103)	242	2086	80	84	163 (140)	588	6965			
PLOC _{r=25}	141	107	151 (129)	438	4718	110	183	141 (123)	261	2060	78	88	210 (174)	611	6699			
PLOC _{r=100}	138	111	287 (262)	561	4671	110	187	259 (241)	377	2138	77	85	375 (338)	793	7081			

TABLE 1

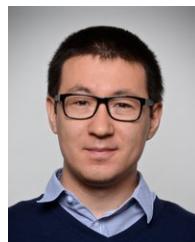
Performance comparison of the tested methods. The reported numbers are averaged over three different viewpoints for each scene. The best results are highlighted in bold. For computing the SAH cost, we used $c_T = 3$ and $c_I = 2$. Build times in parentheses correspond to kernel times.

	Conference	Happy Buddha	Soda Hall	Hairball	Manuscript	Pompeii	San Miguel	Vienna	Power Plant									
	SAH build cost [-]	SAH build time [ms]	SAH build cost [-]	SAH build time [ms]	SAH build cost [-]	SAH build time [ms]	SAH build cost [-]	SAH build time [ms]	SAH build cost [-]									
AAC-Fast	84	18	178	74	179	125	1135	205	98	241	179	324	140	462	113	481	164	669
AAC-HQ	84	56	179	203	177	377	1112	543	103	751	171	1019	137	1523	109	1607	215	2620
PLOC _{r=10}	85	19	175	36	179	43	1092	63	91	62	175	89	144	128	111	119	80	163
PLOC _{r=25}	84	22	175	39	176	50	1085	76	96	77	172	105	141	151	110	141	78	210
PLOC _{r=100}	85	31	179	63	177	88	1081	124	101	134	170	203	138	287	110	259	77	375

TABLE 2

Comparison of AAC by Gu et al. [3] and our method. We report two different quality settings for AAC (AAC-Fast, AAC-HQ) and three settings for PLOC. The table shows the SAH cost and the corresponding build times. AAC was measured on a CPU with 4 physical cores.

- [4] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," Tech. Rep., 1966.
- [5] L. R. Domingues and H. Pedrini, "Bounding Volume Hierarchy Optimization through Agglomerative Treelet Restructuring," in *Proceedings of High-Performance Graphics*, 2015, pp. 13–20.
- [6] S. M. Rubin and T. Whitted, "A 3-dimensional Representation for Fast Rendering of Complex Scenes," *SIGGRAPH Comput. Graph.*, vol. 14, no. 3, pp. 110–116, Jul. 1980.
- [7] H. Weghorst, G. Hooper, and D. P. Greenberg, "Improved Computational Methods for Ray Tracing," *ACM Transactions on Graphics*, vol. 3, no. 1, pp. 52–69, Jan. 1984.
- [8] T. L. Kay and J. T. Kajiya, "Ray Tracing Complex Scenes," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 269–278, Aug. 1986.
- [9] J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Comput. Graph. Appl.*, vol. 7, no. 5, pp. 14–20, May 1987.
- [10] V. Havran, R. Herzog, and H.-P. Seidel, "On the Fast Construction of Spatial Data Structures for Ray Tracing," in *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, Sept 2006, pp. 71–80.
- [11] I. Wald, "On Fast Construction of SAH-based Bounding Volume Hierarchies," in *Proceedings of Symposium on Interactive Ray Tracing*, 2007, pp. 33–40.
- [12] I. Wald, S. Boulos, and P. Shirley, "Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies," *ACM Trans. Graph.*, vol. 26, no. 1, Jan. 2007.
- [13] T. Ize, I. Wald, and S. G. Parker, "Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures," in *Proceedings of Symposium on Parallel Graphics and Visualization*, 2007, pp. 101–108.
- [14] W. Hunt, W. R. Mark, and D. Fussell, "Fast and Lazy Build of Acceleration Structures from Scene Hierarchies," in *Proceedings of Symposium on Interactive Ray Tracing*, Sept 2007, pp. 47–54.
- [15] M. J. Doyle, C. Fowler, and M. Manzke, "A Hardware Unit for Fast SAH-optimised BVH Construction," *ACM Trans. Graph.*, vol. 32, no. 4, pp. 139:1–139:10, Jul. 2013.
- [16] K. Ng and B. Trifonov, "Automatic Bounding Volume Hierarchy Generation Using Stochastic Search Methods," in *CPSC532D Mini-Workshop "Stochastic Search Algorithms"*, April 2003.
- [17] A. Kensler, "Tree Rotations for Improving Bounding Volume Hierarchies," in *Proceedings of Symposium on Interactive Ray Tracing*, 2008, pp. 73–76.
- [18] J. Bittner, M. Hapala, and V. Havran, "Fast Insertion-Based Optimization of Bounding Volume Hierarchies," *Computer Graphics Forum*, vol. 32, no. 1, pp. 85–100, 2013.
- [19] T. Karras and T. Aila, "Fast Parallel Construction of High-Quality Bounding Volume Hierarchies," in *Proceedings of High Performance Graphics*. ACM, 2013, pp. 89–100.
- [20] T. Aila, T. Karras, and S. Laine, "On Quality Metrics of Bounding Volume Hierarchies," in *Proceedings of High Performance Graphics*. ACM, 2013, pp. 101–108.
- [21] H. Dammertz, J. Hanika, and A. Keller, "Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays," *Computer Graphics Forum*, vol. 27, pp. 1225–1233(9), 2008.
- [22] I. Wald, C. Benthin, and S. Boulos, "Getting rid of packets - Efficient SIMD single-ray traversal using multi-branching BVHs -," in *Symposium on Interactive Ray Tracing*, 2008, pp. 49–57.
- [23] M. Ernst and G. Greiner, "Multi bounding volume hierarchies," in *Symposium on Interactive Ray Tracing*, 2008, pp. 35–40.
- [24] J. A. Tsakok, "Faster Incoherent Rays: Multi-BVH Ray Stream Tracing," in *Proceedings of High Performance Graphics*, 2009, pp. 151–158.
- [25] M. Ernst and G. Greiner, "Early Split Clipping for Bounding Volume Hierarchies," in *Symposium on Interactive Ray Tracing*, 2007, pp. 73–78.
- [26] S. Popov, I. Georgiev, R. Dimov, and P. Slusallek, "Object partitioning considered harmful: Space subdivision for bvhs," in *Proceedings of High Performance Graphics*, 2009, pp. 15–22.
- [27] M. Stich, H. Friedrich, and A. Dietrich, "Spatial Splits in Bounding Volume Hierarchies," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 7–13.
- [28] P. Ganestam and M. Doggett, "SAH Guided Spatial Split Partitioning for Fast BVH Construction," *Comp. Graphics Forum*, 2016.
- [29] V. Fueterling, C. Lojewski, F.-J. Pfreundt, and A. Ebert, "Parallel Spatial Splits in Bounding Volume Hierarchies," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2016, pp. 21–30.
- [30] C. Wächter and A. Keller, "Instant Ray Tracing: The Bounding Interval Hierarchy," in *Proceedings Eurographics Symposium on Rendering*, 2006, pp. 139–149.
- [31] M. Eisemann, C. Woizischke, and M. Magnor, "Ray Tracing with the Single Slab Hierarchy," in *VMV*, 2008, pp. 373–381.
- [32] Y. Gu, Y. He, and G. E. Blelloch, "Ray Specialized Contraction on Bounding Volume Hierarchies," *Computer Graphics Forum*, 2015.
- [33] I. Wald, "Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 1, pp. 47–57, 2012.
- [34] P. Ganestam, R. Barringer, M. Doggett, and T. Akenine-Möller, "Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees," *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 3, pp. 23–42, 2015.
- [35] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Comput. Graph. Forum*, vol. 28, no. 2, pp. 375–384, 2009.
- [36] J. Pantaleoni and D. Luebke, "HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry," in *Proceedings of High Performance Graphics*, 2010, pp. 87–95.
- [37] K. Garanzha, J. Pantaleoni, and D. McAllister, "Simpler and Faster HLBVH with Work Queues," in *Proceedings of High Performance Graphics*, 2011, pp. 59–64.
- [38] M. Vinkler, J. Bittner, V. Havran, and M. Hapala, "Massively Parallel Hierarchical Scene Processing with Applications in Rendering," *Computer Graphics Forum*, vol. 32, no. 8, pp. 13–25, 2013.
- [39] T. Karras, "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees," in *Proceedings of High Performance Graphics*, 2012, pp. 33–37.
- [40] C. Apetrei, "Fast and Simple Agglomerative LBVH Construction," in *Computer Graphics and Visual Computing (CGVC)*, 2014.
- [41] D. Meister and J. Bittner, "Parallel BVH Construction using k-means Clustering," *The Visual Computer (Proceedings of Computer Graphics International)*, 2016.
- [42] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.
- [43] D. Merrill and A. Grimshaw, "High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.
- [44] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of High Performance Graphics*, 2009, pp. 145–149.



Daniel Meister is a Ph.D. candidate at the Czech Technical University in Prague. His research interests include data structures for ray tracing, GPGPU, and parallel computing.



Jiří Bittner is an associate professor at the Department of Computer Graphics and Interaction of the Czech Technical University in Prague. He received his Ph.D. in 2003 from the same institution. For several years he worked as a researcher at the Vienna University of Technology. His research interests include visibility computations, real-time rendering, spatial data structures, and global illumination. He participated in a number of national and international research projects and also several commercial projects dealing with real-time rendering of complex scenes.