EUROGRAPHICS 2018 / D. Gutierrez and A. Sheffer
(Guest Editors)

Volume 37 (2018), Number 2

Parallel Reinsertion for Bounding Volume Hierarchy Optimization

D. Meister and J. Bittner

Czech Technical University in Prague, Faculty of Electrical Engineering

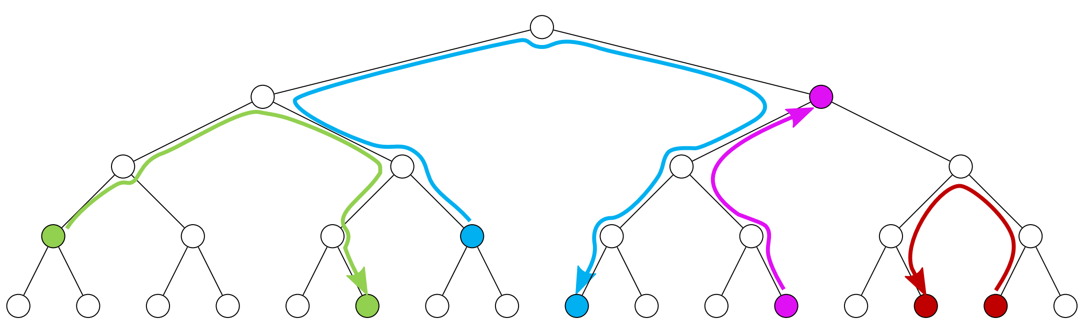

Figure 1: *An illustration of our parallel insertion-based BVH optimization. For each node (the input node), we search for the best position leading to the highest reduction of the SAH cost for reinsertion (the output node) in parallel. The input and output nodes together with the corresponding path are highlighted with the same color. Notice that some nodes might be shared by multiple paths. We move the nodes to the new positions in parallel while taking care of potential conflicts of these operations.*

Abstract

We present a novel highly parallel method for optimizing bounding volume hierarchies (BVH) targeting contemporary GPU architectures. The core of our method is based on the insertion-based BVH optimization that is known to achieve excellent results in terms of the SAH cost. The original algorithm is, however, inherently sequential: no efficient parallel version of the method exists, which limits its practical utility. We reformulate the algorithm while exploiting the observation that there is no need to remove the nodes from the BVH prior to finding their optimized positions in the tree. We can search for the optimized positions for all nodes in parallel while simultaneously tracking the corresponding SAH cost reduction. We update in parallel all nodes for which better position was found while efficiently handling potential conflicts during these updates. We implemented our algorithm in CUDA and evaluated the resulting BVH in the context of the GPU ray tracing. The results indicate that the method is able to achieve the best ray traversal performance among the state of the art GPU-based BVH construction methods.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Raytracing—I.3.5 [Computer Graphics]: Object Hierarchies—I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

1. Introduction

Ray tracing is the underlying engine of many contemporary image synthesis algorithms. An elementary operation of ray tracing is to find the nearest intersection with a scene for a given ray. The major bottleneck is the time complexity which is linear in the number of scene primitives for a näive approach. In practice, millions of rays are tested against millions of scene primitives; the näive approach becomes practically inapplicable. The complexity issue motivated researchers to arrange the scene into various spatial data structures which accelerate ray tracing by orders of magnitude.

Nowadays, *bounding volume hierarchy* (BVH) is one of the most popular acceleration spatial data structure. In the terminology of graph theory, the BVH is a rooted tree containing references to scene primitives in leaves and bounding volumes in interior nodes. The bounding volumes tightly enclose the scene primitives in the corresponding subtree. In the context of ray tracing, we typically use binary or quaternary BVH with axis-aligned bounding boxes. For a given scene, the space of valid BVHs suffers from combinatorial explosion. We can express the quality of the resulting BVH in terms of the SAH cost [MB90]. The problem of finding an SAH-optimal BVH is believed to be NP-hard [KA13]. Due to the hardness, various heuristics are employed to construct a good BVH.

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.

High-quality BVHs are desirable in both online and offline rendering. In offline rendering, increasing BVH quality might save hours or even days of computational time. In online rendering, a scene consists of static and dynamic geometry. The static part is the same in every frame, and thus it pays off to precompute a high-quality BVH for it. The insertion-based optimization method proposed by Bittner et al. [BHH13] achieves the best results in terms of the BVH quality up to date [AKL13]. The method iteratively selects a BVH node and removes it, then traverses the BVH from the root and looks for the best position for the insertion using branch-and-bound pruning approach with the priority queue. The quality, however, comes at a cost: the sequential CPU-based BVH optimization requires computational times several orders of magnitude higher than the recent GPU-based BVH builders.

In this paper, we propose a new highly parallel algorithm based on reinsertions. Our principal idea is that we do not have to remove the node from the BVH to compute the SAH cost reduction yielding from its reinsertion. Unlike the original algorithm, we traverse the BVH from many nodes in parallel while simultaneously tracking the SAH cost reduction. We use the original position of each node as a lower bound of the SAH cost reduction to efficiently prune the search. The traversal is GPU-friendly as we do not use any priority queue or another auxiliary data structure. Using this traversal, we are able to process all nodes in parallel. In the second phase, the nodes are moved in the tree in parallel. Conflicts between nodes occur and must be resolved. We use a greedy approach prioritizing nodes with the higher SAH cost reduction based on atomic locks. The method optimizes a BVH iteratively by reinserting batch of nodes in each iteration and reducing the SAH cost until it converges to a local minimum.

The algorithm is designed to be highly parallel exploiting computational power of modern many-core architectures, and thus execution times are just fractions of times of the original reinsertion method [BHH13]. In many cases, our algorithm also reaches slightly better results in terms of quality since we search for the best reinsertion position for all nodes and use looser termination criteria. The algorithm can be easily plugged into existing GPU-based ray tracing frameworks as it can be used as an optional BVH post-processing if maximum ray traversal performance is required.

2. Related Work

The bounding volume hierarchy has a long history dating back to 1980s. Rubin and Whitted [RW80] were the first ones who used manually created BVHs in rendering. Kay and Kajiya [KK86] designed the construction algorithm using spatial median splits. Goldsmith and Salmon [GS87] proposed a metric which estimates the efficiency of BVHs today known as *surface area heuristic* (SAH). The SAH metric is often associated with the full-sweep SAH algorithm, which recursively splits scene primitives by axis-aligned planes into left and right subtree based on the SAH metric. The evaluation of all splitting planes is rather costly, and thus Havran et al. [HHS06], Wald et al. [Wal07, WBS07], and Ize et al. [IWP07] proposed an approximate SAH evaluation based on the concept of binning. The full-sweep algorithm was a de facto reference algorithm for a long time. Today, there are several ways how to do better than the full-sweep algorithm. Walter et al. [WBKP08] proposed to construct BVH bottom-up by agglomerative clustering. Kensler [Ken08] optimizes an existing BVH by tree rotations. Bittner et al. [BHH13] also optimizes an existing BVH by more general remove-and-insert operations. Another popular concept is sorting scene primitives along the Morton curve [Mor66] which not only coherently fills the space but also implicitly encodes the hierarchy using spatial median splits. This approach was recently extended by Vinkler et al. [VBH17] by taking into account also sizes of scene primitives.

Nowadays scenes are more and more complex, and there is also a tendency to use ray tracing in interactive applications with dynamic content. Thus, not only the quality but also the construction speed became an important aspect. With hardware development, researchers started to design the construction algorithms to utilize parallel capabilities of both multi-core CPUs and many-core GPUs.

Multi-core CPU Gu et al. [GHFB13] proposed parallel approximate agglomerative clustering using the Morton curve to partition scene primitives into coherent clusters. Ganestam et al. [GBDAM15] proposed a similar approach using SAH splits instead of agglomerative clustering. Recently, Hendrich et al. [HMB17] revisited an idea of Hunt et al. [HMF07] to use an existing hierarchy to accelerate the construction. Benthin et al. [BWWA17] used the same idea to improve the concept of two-level BVHs.

Many-core GPU Lauterbach et al. [LGS*09] proposed an algorithm known as LBVH using the concept of the Morton codes and spatial median splits. Karras [Kar12] and Apetrei [Ape14] improved the LBVH method using the concept of radix trees. The LBVH method is the fastest construction algorithm up to date but resulting BVHs suffer from lack of quality as the method employs only spatial median splits. To improve this weakness, Pantaleoni and Luebke [PL10], Garanzha et al. [GPM11] extended the LBVH method into a method known as HLBVH which employs SAH splits for the top levels of the hierarchy. Karras and Aila [KA13] proposed an optimization method by parallel subtree restructuring. Domingues and Pedrini [DP15] further extended this method by employing agglomerative clustering instead of the brute force algorithm. Recently, Meister and Bittner [MB17] proposed an efficient method based on parallel locally-ordered agglomerative clustering. Recently, Ylitie et al. [YKL17] showed that excellent ray traversal performance can be achieved by using compressed wide BVHs.

3. Cost Model

The surface area heuristic (SAH) [GS87, MB90] expresses the expected number of operations for finding the nearest intersection for a given BVH and a random ray. The cost of a BVH node is given by the recurrence equation:

$$c(N) = \begin{cases} c_T + \frac{SA(N_L)}{SA(N)} c(N_L) + \frac{SA(N_R)}{SA(N)} c(N_R) & \text{if } N \text{ is interior node,} \\ c_I |N| & \text{otherwise,} \end{cases} \tag{1}$$

where $c(N)$ is the cost of the subtree with a root N, $SA(N)$ is the surface area of bounding box of the node N, N_L and N_R are left and right children of the node N, respectively, and $|N|$ is the number of

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

triangles in the node N. The constants c_T and c_I express the average cost of the traversal step and ray-triangle intersection, respectively.

The underlying assumptions of SAH are that the distribution of rays is uniform and that the rays are unoccluded. Under these assumptions, the ratio of surface area of child node and parent node is equal to the conditional probability of hitting a child node when the parent node is hit. We can rewrite Equation 1 by unrolling the recurrence:

$$c(N) = \frac{1}{SA(N)}\left[c_T \sum_{N_i} SA(N_i) + c_I \sum_{N_l} SA(N_l)|N_l|\right], \qquad (2)$$

where N_i and N_l denote interior and leaf nodes of the subtree with root N, respectively. Although the assumptions of this cost model are generally not met, it works very well in practice.

4. Bounding Volume Hierarchy Optimization

We propose an algorithm based on parallel reinsertion (PRBVH) for optimization of bounding volume hierarchies. First, we define the reinsertion operation. Then, we describe the two main ideas how to independently search for the best position for the reinsertion and how to resolve conflicts between nodes. Last, we put these ideas together and provide a brief description of the optimization algorithm.

4.1. Reinsertion Operation

For a given node (*input node*), we search for another node (*output node*), such that the insertion of the input node at the output node reduces the SAH cost. The *reinsertion* is a combination of a removal and an insertion. We remove the input node (with the whole subtree) from the tree together with its parent, and we connect its sibling to the original position of the parent (removal). Then, we insert the input node into the output node using the parent as a common parent (insertion). An example of the reinsertion operation is depicted in Figure 2.

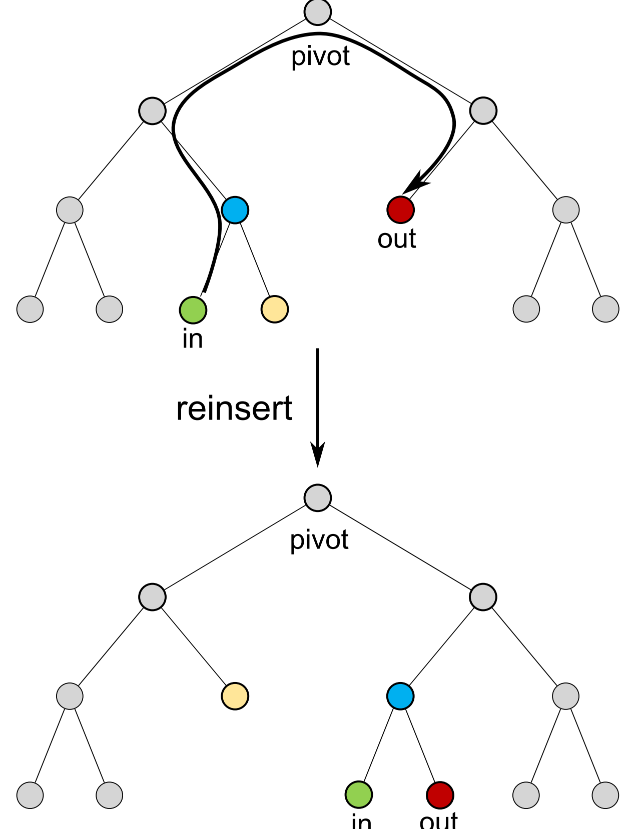

Figure 2: *An illustration of the reinsertion operation of the green node (the input node) into the red node (the output node). Suppose that the green node knows the path to the red node. First, we remove the green node together with the blue node (its parent) from the tree. Then we place the yellow node (its sibling) into the original position of the blue node. Last, we insert the green into the red node using the blue node as a common parent.*

4.2. Parallel Search

The goal of the optimization is to reduce the SAH cost. If we use one triangle per leaf, then all terms in Equation 2 become constant except the surface areas of interior bounding boxes $SA(N_i)$. In other words, the SAH cost becomes directly proportional to the sum of surface areas of bounding boxes. Using one triangle per leaf during the optimization is beneficial as the optimization is less constrained. We formulate our problem as maximization of the surface area decrease which corresponds to the SAH cost reduction. The surface area decrease is equal to the sum of decreases of surface areas of affected bounding boxes.

Notice that only bounding boxes on the path between the input and output nodes might be affected. This path starts by traversing tree up to the root and then at some point it breaks into a sibling subtree. We denote the node where the path breaks as the *pivot node* (see Figure 2). The surface area decrease for all nodes on the path before the pivot node is non-negative (corresponding to the removal), and the surface area decrease on the path after the pivot node is non-positive (corresponding to the insertion). More precisely, we can decompose the path into three zones based on the sign of the surface area decrease: positive zone, zero zone, and negative zone. The positive zone starts at the input node and ends at some node before the pivot node. The zero zone always contains the pivot node since the bounding box of the pivot node is never affected. The negative zone starts at some node behind the pivot node and ends at the output node. The decomposition into the zones is depicted in Figure 3.

Let us look closer how to find the output node. We want to traverse the tree from a given input node to all other nodes (except the nodes below the input node). For the traversal, we want to avoid using a stack or another data auxiliary data structure. To traverse all nodes, we proceed from the input node visiting all siblings' subtrees on the way up to the root. We can visit siblings' subtree simply by pre-order (or post-order) traversal without any auxiliary data structure using just parent links. We have two states indica-

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

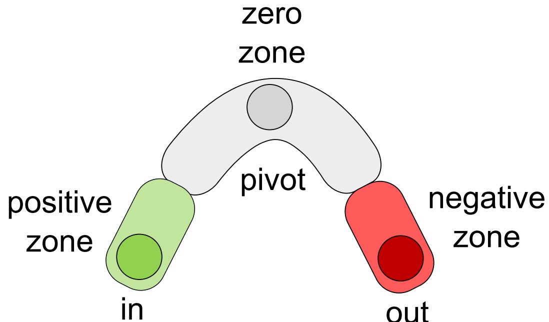

Figure 3: *An illustration of the path between the input and output nodes decomposed into three zones based on the sign of the surface area decrease of the corresponding node: positive zone (green), zero zone (gray), and negative zone (red).*

ting whether we are in the down or up phase of the traversal. In the down phase, we go to the left child if the current node is interior. Otherwise, we switch to the up phase. In the up phase, if we came from the left child, then we go to the right child and switch to the down phase. Otherwise, we go to the parent.

We accumulate the surface area decrease from the input node to the current output node. The accumulated surface area decrease is the sum of the individual surface area decreases of the nodes on the path from the input node to the current output node excluding the input node, its parent, and the current output node. We use the parent as a common parent for the input and output nodes, and thus its surface area decrease is the difference between the surface area of the parent node and the surface area of the union the input and output nodes.

During the down phase, we apply pruning as visiting all nodes would be inefficient. We want to estimate the upper bound of the surface area decrease of the parent node, which is equivalent to estimating the lower bound of the surface area of the union of the input and the output nodes since the surface area of the parent is known. The surface area of the union must be greater than or equal to the surface area of the input node. Thus, if the accumulated surface area decrease plus the upper bound is less than or equal to the best surface area decrease found so far, then we can prune the search as the surface area decrease on the way down is non-positive.

4.2.1. Search Algorithm Details

Now let us look into the details of the procedure which is given in Algorithm 1. We fetch the bounding box of the input node ($\mathbf{b}_{in}$) and its parent ($\mathbf{b}_{parent}$) as we need them to compute the surface area decrease (lines 1–2). We want to visit all siblings on the way up to the root. The sibling of the input node is the first sibling we want to visit. Thus, we set the sibling as the current output node (*out*), the parent of the input node as the current pivot node (*pivot*), and the down flag (*down*) to true (lines 6–8). The accumulated surface area decrease (d) is the sum of surface area decreases from the input node to the current output node excluding the input node, its parent, and the current output node; and it is initially set to 0 (line 5). The difference between the surface areas of the parent node and the input node (d_{bound}) is the upper bound of the surface area decrease of the parent node (line 4), we use it to prune the search. The surface area decrease of a node on the path up to the pivot node is the original surface area of the node minus the surface area of the node after the removal of the input node. The surface area after the removal is equal to the surface area of the union of bounding boxes of all siblings up to the node. We can compute this union of bounding boxes ($\mathbf{b}_{pivot}$) incrementally as we update the pivot node, and initially it is set to the empty box (line 3). We also set the sibling as the best output node found so far (out_{best}) and the best surface area decrease so far (d_{best}) to 0 which corresponds to the original position (lines 5–6).

After the initialization, we enter the main loop (lines 9–54). We fetch the bounding box of the current output node and compute the union of the bounding boxes of the input and output nodes (lines 10–11).

The main loop consists of three different cases based on the state of the search: the down phase (lines 13–23) and the up phase (lines 46–51) of the pre-order traversal, and the case when the pre-order traversal of a subtree is finished (lines 27–44).

If the down flag is true, we enter the down phase. We compute the surface area decrease for the current output node, which is the sum of the accumulated surface area decrease and the surface area decrease of the parent node. Eventually, we update the best output node found so far (lines 14–17). We update the accumulated surface area decrease as we want to continue down (line 18). At this point, we apply the pruning we discussed above (lines 19–21). If the upper bound of the parent node plus the accumulated surface area decrease is less or equal to the best surface area decrease found so far, we prune the search. Thus, if the current output node is a leaf or the search was pruned, we set the down flag to false. Otherwise, we continue to the left child.

If the down flag is false, then the search was either pruned, or a leaf node was reached. We have to subtract the surface area decrease of the current output node (line 25). There are two cases: the first case is the situation when we finish the pre-order traversal of the subtree, the second case is the up phase of the pre-order traversal.

In the first case, the current output node is a child node of the pivot node. We update the pivot bounding box by merging it with the current output node, and we switch to the pivot node (lines 27–29). We compute the surface area decrease for the pivot node similarly to the down phase, and eventually, we update the best output node found so far (lines 33–36). We update the accumulated surface area decrease (line 37). Note that we skip the parent node of the input node because such insertion does not make sense. If we reach the root, we are done (lines 39–41). Otherwise, we update the pivot node by switching it to its parent and continue to its sibling (lines 41–44).

In the second case, we enter the up phase of the pre-order traversal. If we come from the left subtree, we switch to the right subtree (lines 47–48). Otherwise, we continue to the parent node (line 50).

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

4.3. Parallel Reinsertion

The procedure described above is able to find the optimized position for all nodes in parallel. The second phase of the algorithm is to move the nodes to their new positions. In this step, each input node should be removed from its original location and then reinserted into to the position indicated by the output node. More precisely, it should become a sibling of the output node.

However, the reinsertion operations of multiple nodes can be in mutual conflicts. We can distinguish between two types of conflicts: (1) *topological conflicts* and (2) *path conflicts*.

4.3.1. Topological Conflicts

The topological conflicts are caused by two or more reinsertion operations trying to modify the topology of the same node in the tree. The topological changes induced by the reinsertion operation are localized in the proximity of the input and output nodes. More precisely, the topological conflicts involve six nodes in total. These are the nodes participating in the removal (the input node, its sibling, its parent, and its grandparent) and the insertion (the output node and its parent) (see Figure 2).

To resolve the conflict, we aim to lock the nodes involving the topological change to prevent race conditions. We use a greedy locking scheme based on atomic operations: if any two reinsertions share the same node involving a topological change, then the reinsertion with the higher surface area decrease locks the node. Prior to reinsertion, we verify whether all nodes which aimed to be locked by the operation were locked successfully. If this is the case, the reinsertion is performed. Otherwise, the reinsertion is abandoned as it was in conflict with another reinsertion operation that yielded higher SAH cost reduction.

4.3.2. Path Conflicts

Additional conflicts arise if multiple reinsertions share nodes on the paths and try to modify the bounding boxes of shared nodes in a different way. Recall that each path consists of up to three zones which induce different modification of the corresponding bounding boxes (as it was shown in Figure 3). If such types of conflicts occur, then the total surface area decrease in the shared part is not simply the sum of the surface area decreases of the individual reinsertions. In general, some of these path conflicts are harmless (e.g. conflicts of nodes from the zero zone), some may support each other in terms of the SAH cost reduction (e.g. adding overlapping areas in the negative zone), and some work against each other (e.g. conflict of positive and negative zones). The path conflicts can be resolved by locking all nodes on the reinsertion paths in the same way as topology nodes. Note that to access all nodes on the path, we encode the path into the bit set already during the search phase. In this way, all nodes on the path can be accessed during the locking and lock verification phases.

4.3.3. Locking Strategy

For predictable SAH cost reduction, we should combine both above-described locking strategies. We call this combined locking strategy (resolving both topological and path conflicts) *conservative*. Using the conservative strategy, the algorithm is guaranteed to converge as the total surface area decrease is the sum of individual surface area decreases for all successfully locked paths.

However, for the correctness of the proposed algorithm in terms of valid BVH topology it suffices to resolve only the topological conflicts. We call this locking strategy *aggressive*. With the aggressive strategy, the total surface area decrease will generally not correspond to the sum of individual surface area decreases: it can either be lower or even higher (if multiple operations support each other).

Experimentally, we have observed that the aggressive strategy leads to faster BVH convergence; moreover, it converges to a BVH with slightly lower SAH cost than the conservative one. At first sight, this was a surprising result, and therefore we provide a more detailed analysis of the parallel reinsertion in the next section.

4.3.4. Superiority of the Aggressive Strategy

Suppose that we have a set of bounding boxes enclosed by their parent bounding box. Let us investigate how many bounding boxes from the set define the parent bounding box. Bounding boxes strictly inside the parent box can be excluded, i.e. they do not define the boundary of the bounding box. If two bounding boxes touch a face, we can also exclude one of these bounding boxes supposing that the bounding box does not define another face. All in all, at most six bounding boxes define the parent bounding box in 3D, i.e. one for each face.

Let us further investigate what might happen after removal and insertion of multiple nodes into a single node. We consider first only the removal of multiple nodes. Only nodes defining the original bounding box might have positive surface area decrease. If two nodes define the same face, then both nodes have zero surface area decrease (at least for that particular face). Other nodes are strictly inside the original bounding box, and thus they also have zero surface area decrease. If we remove the nodes with positive surface area decrease, it might happen that the total surface area decrease is less than the sum of the individual surface area decreases (the decreases are shared among multiple reinsertion paths). If two faces defined by two nodes share an edge, then the surface area decrease of the region around the edge is counted only once, even if it is included in both individual surface area decreases. Let us remove these nodes and update the bounding box. We can further shrink bounding box by removing the rest of the nodes. These nodes originally assumed zero surface area decrease, and thus their removal can only be beneficial in terms of the overall SAH cost. An example of the removal of multiple nodes is depicted in Figure 4.

A very similar situation arises with the insertion of multiple nodes. We suppose that all nodes have negative surface area decrease as the bounding box should be enlarged by the insertion. Some of these nodes define the final bounding box. If two nodes define the same face of the final bounding box, then the shared insertions are beneficial as the bounding box is enlarged only once (after the first insertion). On the other hand, when we insert the nodes defining the final bounding box, it might happen that the total surface area decrease is lower than the sum of the individual surface area decreases. If two faces defined by two nodes share an edge, then the total surface area decrease is the sum of the individual surface area decreases plus the (negative) surface area decrease induced by the

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

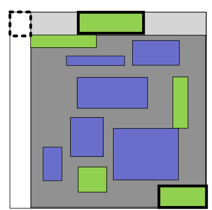
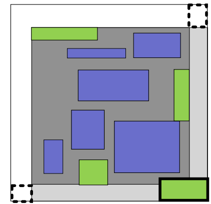
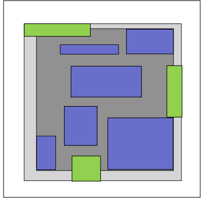
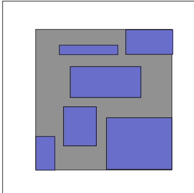

Figure 4: *An illustration of the removal of multiple nodes (green) from a single node. The nodes defining the original bounding box (bold border) yield a positive surface area decrease, the other nodes are expected to yield zero surface area decrease. We successively remove three nodes defining the bounding box. In this case, the shared area (dashed border) is counted only once instead of twice, and thus the actual total surface area decrease is reduced by half of the shared area. On the other hand, the combined surface area decrease becomes larger than expected by removing the nodes that did not define the boundaries of the node bounding box and which originally expected zero surface area decrease.*

region around the edge. Let us insert these nodes and update the bounding box. We will not enlarge the bounding box by inserting the rest of the nodes since we already inserted the nodes defining the final bounding box. These nodes have negative surface area decrease, and thus the shared insertion is beneficial in terms of the overall SAH cost. An example of the insertion of multiple nodes is depicted in Figure 5.

Let us visualize the worst and best case of the total surface area decrease in a node shared by two paths. In the best case of the removal, the bounding box might completely collapse into a degenerated bounding box with zero surface area by removing multiple nodes. However, the bounding box will not change by removing any single node. In the worst case of the removal, the bounding box might collapse into degenerated bounding box with zero surface area by removing a single node, and the surface area decreases from other nodes are not counted at all. These two cases are depicted in Figure 6 (left). In the best case of insertion, the bounding box will be enlarged just by one node and the others will not change it at all. In the worst case of the insertion, imagine the situation that we insert two degenerated bounding boxes (points) into a degenerated bounding box (point). We enlarge the bounding box from a point to a line segment still with zero surface area by applying any individual insertion. However, by inserting both nodes, the bounding box will be enlarged by the combined effect of both inserted nodes. These latter two cases are depicted in Figure 6 (right).

Let us sum up why the aggressive strategy performs better than the conservative one. In the aggressive strategy, only nodes participating in topology changes are locked, and thus significantly more reinsertions can be performed in parallel in a given iteration. Especially, at the beginning, the positive surface area decreases are much greater than negative surface area decreases, and thus the combined effect of multiple reinsertions is beneficial. Additionally, there are typically much more nodes to be removed not defining the original bounding box and much more nodes to be inserted not defining the final bounding box for which the shared removal and insertion combines positively as discussed above.

4.4. Complete Algorithm

The algorithm takes an arbitrary BVH as an input and optimizes it iteratively. In each iteration, a batch of reinsertion operations is performed. First, each node searches for its best output node in parallel using the search discussed in Section 4.2. Second, the conflicts between nodes are resolved using the locking scheme discussed in Section 4.3. The nodes with successful locks can be reinserted. After the reinsertion, we recompute the bounding boxes and the SAH cost.

4.4.1. Sparse Search

The bottleneck of the algorithm is the search phase. Additionally, when neighboring nodes find their optimized positions, there is a high chance of conflict during the insertion phase. Thus, we introduce an integer parameter $\mu \geq 1$ to control the density of the nodes used for the search. We process only nodes satisfying the following condition:

$$I \pmod{\mu} \equiv i \pmod{\mu}, \tag{3}$$

where i is the index of the node and I the number of the iteration. In other words, we process every μ-th node in round-robin manner based on the number of iteration. By increasing μ, the search gets sparser as fewer nodes are used to initiate the search. Thus, this phase becomes faster, and additionally, there are fewer conflicts during the reinsertion. The optimal setting of the μ parameter is scene dependent, but the results indicate that good results are obtained by using $\mu \in \{4,\ldots,9\}$.

4.4.2. Termination Criteria

In later stages of the optimization, the experiments showed that the total surface area decrease oscillates around zero. In this case, it is reasonable to terminate the optimization. Even if the surface area decrease is still positive but very close to zero, it is beneficial to terminate the optimization earlier which results in a BVH with roughly the same quality as if we had continued. To deal with these issues, we introduce another parameter ε. If the difference between the SAH cost of the previous and the current iterations is less than ε, we either decrement μ by one if it is greater than one or terminate the optimization if it is equal to one. We set the ε parameter to 0.1 which seems to be a reasonable choice according to our experiments.

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

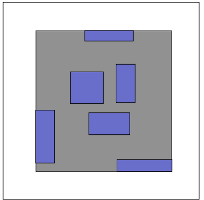
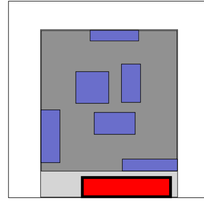
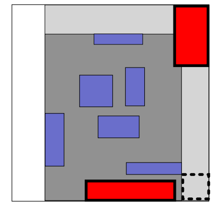
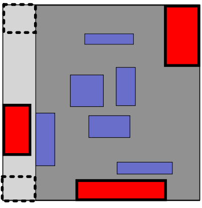
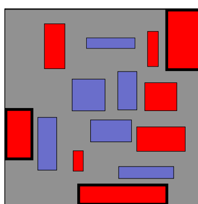

Figure 5: *An illustration of the insertion of multiple nodes (red) into a single node. All nodes assume negative surface area decrease since they enlarge the bounding box of the output node, and thus increasing its surface area. Notice that only three nodes define the final bounding box (bold border). We successively insert these three nodes. The combined surface area increase becomes larger than expected by the extra area caused by multiple insertions (dashed border). On the other hand, we do not have to count the surface area increase caused by the nodes inside the final bounding box (the bounding box was already enlarged by the boundary nodes), which reduces the surface area increase and can easily outweigh the above mentioned unexpected surface area increase.*

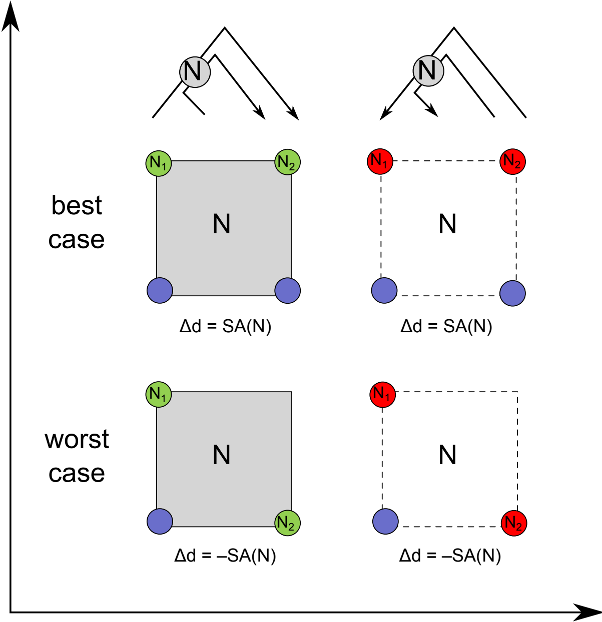

Figure 6: *An illustration of the worst (bottom) and best (top) of the total surface area decrease for two parallel reinsertions N_1 and N_2 in node N: two removals (left) and two insertions (right). For each particular case, we show the difference between the combined total area decrease and the sum of surface area decrease: $\Delta d = d_{1,2} - (d_1 + d_2)$.*

5. GPU Implementation

We implemented our algorithm in CUDA [NBGS08]. We use the structure-of-arrays layout to represent the BVH: left indices (4B), right indices (4B), parent indices (4B), minimal bounds (16B), and maximal bounds (16B). We also use some additional buffers to store intermediate results for each node: surface area decreases (4B), output node indices (4B), and locks (8B). The algorithm takes an arbitrary BVH as an input. The main loop consists of six kernels that are repeatedly executed to optimize the BVH.

Find the best node The first kernel implements the search procedure (see Algorithm 1) to find the best output node for every node in parallel. We store the surface area decrease and the index of the best output node.

Lock nodes The second kernel implements the locking scheme. We process all nodes with the positive surface area decrease in parallel. We use 64-bit integers to implement the locks: 32 bits for the area decrease and 32 bits for the node index. We use the fact that the result of comparison of positive numbers in the floating point representation is the same as the comparison in the integer representation. Thus, we lock a node by the atomic maximum operation. The reinsertion with the highest surface area decrease always wins the node. We use the node index to avoid deadlocks between two reinsertions with the same surface area decrease prioritizing the node with a higher index. We lock all nodes directly participating in the reinsertion: removal (the input node, its sibling, its parent, and its grandparent) and insertion (the output node and its parent).

Check locks Similarly to the previous kernel, we process all nodes with positive surface area decrease. We check the locks of nodes directly participating in the reinsertion. If any node of the reinsertion was locked by another reinsertion, then the reinsertion will not be performed. In this case, we set the surface area decrease to 0 to exclude these reinsertions for the next step.

Reinsert We check again surface area decreases and perform the reinsertion if it was not excluded in the previous step.

Recompute bounding boxes After the reinsertion, we have to recompute bounding boxes. We use the parallel bottom-up procedure proposed by Karras [Kar12]. Each interior node has a counter initially set to 0. Threads process from leaves up to the root atomically incrementing the counter in interior nodes. If the original value was 0, then the thread is the first one in the node, and thus it is killed. Otherwise, the thread is the second one and continues to the parent.

Compute the SAH cost We compute the SAH cost to adjust the μ parameter or to terminate the optimization. We use the unrolled version of the SAH cost (see Equation 2) which enables computing the SAH cost simply by parallel reduction.

The source codes of the algorithm can be downloaded from the project website: `http://dcgi.felk.cvut.cz/projects/prbvh/`.

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

6. Results and Discussion

We evaluated the PRBVH method using a GPU path tracer based on a high-performance ray tracing kernel of Aila et al. [AL09]. Our path tracing implementation uses next event estimation with two light source samples per hit. We used 128 samples per pixel and 1024×768 image resolution for all measurements. We evaluated all methods using eight publicly available test scenes of various complexity. 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 9.1), Windows 7 OS.

As reference methods, we used the LBVH builder proposed by Karras [Kar12], the ATRBVH builder proposed by Domingues and Pedrini [DP15], the PLOC builder proposed by Meister and Bittner [MB17], and the RBVH method proposed by Bittner et al. [BHH13]. For the LBVH method, we used 60-bit Morton codes. For the ATRBVH method, we used the publicly available implementation of treelet restructuring using treelets of size 20. We modified the implementation to use an adaptive number of iterations. The γ parameter determines how many triangles a treelet must have to be optimized. In the original implementation, the γ parameter is initially set to the treelet size, and it is doubled in each iteration. We optimize the BVH until the γ parameter exceeds the number of triangles in the whole scene. We used the LBVH builder with 60-bit Morton codes as a base for the ATRBVH method. For the PLOC method, we also used a publicly available implementation using the radius set to 100 and 60-bit Morton codes. For the RBVH method, we use BVHs initially built by LBVH and the optimization termination criteria suggested in the original paper.

For our PRBVH method, we exhaustively evaluated the influence of the μ parameter in the range $\{1,\dots,32\}$. We chose three representative settings yielding the best results: PRBVH$^A_{\mu=1}$, PRBVH$^A_{\mu=4}$, and PRBVH$^A_{\mu=9}$. In all cases, we used an adaptive leaf size using the algorithm from the PLOC implementation transforming the structure-of-arrays data layout into the array-of-structures data layout needed by the ray tracing kernel. Different algorithms produce BVHs with different node layout. For the sake of fair comparison, we modified the algorithm to transform BVHs into the GPU-friendly BFS layout.

The results are summarized in Table 1. For each tested 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 (overall and for specific ray types), and the total build time (including both the build time and the optimization time). For the build time, we report the sum of kernel times for the GPU-based methods, and CPU times for the RBVH method. For all PRBVH tests, we optimize BVHs initially constructed by LBVH with 60-bit Morton codes. The reported trace speed is the average of three different representative camera views to reduce the influence of view dependency.

We can see that both methods PRBVH and RBVH converge to very similar SAH costs and achieve the best results overall. Particularly, the PRBVH method reaches up to 67% lower SAH costs than LBVH, up to 16% lower SAH costs than ATRBVH, and up to 22% lower SAH costs than PLOC.

The PRBVH and RBVH methods achieve the highest trace speed, while PRBVH yields slightly higher trace speed than RBVH in six of the eight tested scenes. Particularly, the PRBVH method reaches up to 114% higher trace speed than LBVH, up to 12% higher trace speed than ATRBVH, and up to 31% higher trace speed than PLOC. Notice that all settings of the PRBVH method independently of the parameter μ yield very similar trace speed and SAH costs. The LBVH method, which achieves the best build times overall, is up to two orders of magnitude faster than the PRBVH method. The ATRBVH and PLOC methods are up to one order of magnitude faster than the PRBVH method. However, the PRBVH method is still almost two orders of magnitude faster than the RBVH method.

We compared the progress of optimization of BVHs initially built by LBVH and ATRBVH. In general, optimization times when starting from ATRBVH are lower than when starting from LBVH. The SAH costs are roughly the same in both cases. The comparison for the Manuscript scene is given in Figure 8. In this particular case, the optimization of LBVH converges to lower SAH cost than the optimization of ATRBVH, and the convergence is faster. We also compared the aggressive and conservative strategies. In all cases, the aggressive strategy converges faster to lower SAH costs. The comparison for the Crown scene is given in Figure 7. Notice the overhead in the locking and checking phase caused by the more complicated implementation of the conservative strategy.

From Figures 7 and 8, we can see two additional remarks. First, the major bottleneck is the search phase (even if it is accelerated by the μ parameter). Second, we can observe the exponential tendency in the SAH cost reduction in time. In other words, if we want to reduce the SAH cost further, we need more and more time.

7. Conclusion and Future Work

We proposed a new parallel method for BVH optimization that targets contemporary GPU architectures. The method is based on the idea of iterative application of reinsertions [BHH13]. We reformulated the search phase of the algorithm while exploiting the observation that there is no need to remove the nodes from the BVH prior to finding their optimized positions in the tree. This allowed us to apply the search for optimized node positions in the BVH in a massively parallel fashion. We designed an algorithm for updating all nodes in parallel while handling potential conflicts either by conservative or aggressive strategy. We also provide the first deeper analysis of the influence of concurrent update operations on the BVH. We implemented our algorithm in CUDA and made the implementation publicly available. The evaluation in the context of the GPU ray tracing shows that the proposed method is able to achieve the best ray traversal performance among the state of the art GPU-based BVH construction methods making it a good candidate for GPU-based high-quality renderers. On the tested scenes the proposed technique achieves ray tracing speedup of 4% to 12% w.r.t. ATRBVH and 8% to 31% w.r.t. PLOC.

In the future, we would like to focus on exploiting a priori knowledge about scene modifications within our algorithm. If only a part of the scene is dynamic, the algorithm could exploit temporal coherence by focusing the updates on the BVH nodes corresponding to the moving parts. This is a typical scenario in video games

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

	Happy Buddha	Soda Hall	Hairball	Manuscript	Crown	Pompeii	San Miguel	Vienna
#triangles	1087k	2169k	2880k	4305k	4868k	5632k	7880k	8637k
SAH cost [-]								
LBVH	204	254	1237	182	76	429	251	299
ATRBVH	168	163	1055	95	63	173	136	110
PLOC	179	178	1087	102	67	170	139	110
RBVH	160	**139**	1004	81	63	**158**	**125**	**100**
$PRBVH^{A}_{\mu=1}$	**159**	143	**985**	**80**	**61**	160	128	101
$PRBVH^{A}_{\mu=4}$	**159**	143	**985**	**80**	**61**	159	127	**100**
$PRBVH^{A}_{\mu=9}$	**159**	142	**985**	**80**	**61**	159	126	**100**
Trace speed (Overall) [MRays/s]								
LBVH	110	185	45	164	86	72	62	81
ATRBVH	125	235	51	231	101	116	95	161
PLOC	116	205	42	226	93	113	94	160
RBVH	123	**263**	49	242	93	124	**108**	150
$PRBVH^{A}_{\mu=1}$	**131**	257	54	249	106	123	104	166
$PRBVH^{A}_{\mu=4}$	**131**	256	54	**250**	**107**	123	105	**173**
$PRBVH^{A}_{\mu=9}$	**131**	262	**55**	**250**	**107**	**125**	106	172
Trace speed (Primary / Shadow / Secondary) [MRays/s]								
LBVH	404 / 80 / 56	445 / 256 / 98	92 / 54 / 17	207 / 175 / 93	163 / 108 / 31	111 / 92 / 31	91 / 96 / 28	136 / 96 / 37
ATRBVH	450 / 90 / 64	509 / 322 / 127	98 / 62 / 19	293 / 242 / 131	193 / 126 / 38	161 / 154 / 51	162 / 154 / 41	233 / 196 / 76
PLOC	406 / 86 / 58	464 / 291 / 107	80 / 55 / 15	293 / 238 / 124	167 / 124 / 33	152 / 155 / 49	150 / 156 / 39	241 / 198 / 72
RBVH	460 / 89 / 62	607 / 338 / **150**	93 / 64 / 18	307 / 257 / 136	173 / 119 / 34	175 / **164** / 54	**181 / 171 / 47**	239 / 181 / 69
$PRBVH^{A}_{\mu=1}$	**484 / 93 / 69**	586 / 339 / 143	**115 / 65 / 20**	320 / 260 / **140**	202 / 130 / **40**	175 / 161 / 54	177 / 167 / 45	254 / 200 / 78
$PRBVH^{A}_{\mu=4}$	483 / **93 / 69**	597 / **343** / 141	**115 / 65 / 20**	**322 / 261 / 140**	205 / 131 / **40**	173 / 162 / 54	**181** / 166 / 45	**262 / 207 / 81**
$PRBVH^{A}_{\mu=9}$	479 / **93 / 69**	**614** / 340 / 148	**115 / 65 / 20**	**322 / 261 / 140**	**207 / 132 / 40**	**180 / 164 / 55**	**181** / 168 / 46	**262** / 206 / **81**
Build time [s]								
LBVH	**0.01**	**0.02**	**0.02**	**0.05**	**0.05**	**0.06**	**0.08**	**0.10**
ATRBVH	0.07	0.15	0.18	0.27	0.32	0.42	0.56	0.62
PLOC	0.05	0.08	0.10	0.13	0.19	0.20	0.37	0.29
RBVH	29.41	58.57	91.15	114.00	27.26	207.20	290.82	96.91
$PRBVH^{A}_{\mu=1}$	0.60	2.56	2.96	4.56	3.74	21.08	15.55	26.09
$PRBVH^{A}_{\mu=4}$	0.34	1.23	2.33	1.15	1.67	5.08	6.97	6.72
$PRBVH^{A}_{\mu=9}$	0.42	1.17	3.03	1.14	1.83	4.63	6.37	5.77

Table 1: *Performance comparison of the PRBVH method and reference methods: LBVH [Kar12], ATRBVH [DP15], PLOC [MB17], and RBVH [BHH13]. 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 $c_I = 2$. The PRBVH method corresponds to the aggressive strategy optimizing BVHs built by LBVH.*

where a large part of the scene remains static between successive frames. By focusing the optimization on the dynamic scene parts, we could continuously maintain high-quality BVH in a fraction of the computational time required by the uninformed algorithm.

In terms of theoretical analysis, we see a certain analogy between the insertion-based optimization and 2-OPT, which is a popular heuristic optimizing the solution of the traveling salesman problem. The idea is to take all pairs of edges and swap them to reduce the total length. In the case of the insertion-based optimization, we take all pairs of nodes and try to insert them into each other to reduce the total surface area. There is a well-known generalization of 2-OPT to k-OPT which takes all k-tuple of edges and tries to swap them in all possible ways to reduce the total length and still keep a valid Hamiltonian circle. We can do the same generalization for the insertion-based optimization. Thus, we can take all k-tuples of nodes and try to insert them into each other in all possible ways to reduce the total surface area.

8. Acknowledgements

This research was supported by the Czech Science Foundation under project number GA18-20374S, and the Grant Agency of the Czech Technical University in Prague, grant No. SGS16/237/OHK3/3T/13.

References

[AKL13] AILA T., KARRAS T., LAINE S.: On Quality Metrics of Bounding Volume Hierarchies. In *Proceedings of High Performance Graphics* (2013), ACM, pp. 101–108. 2

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

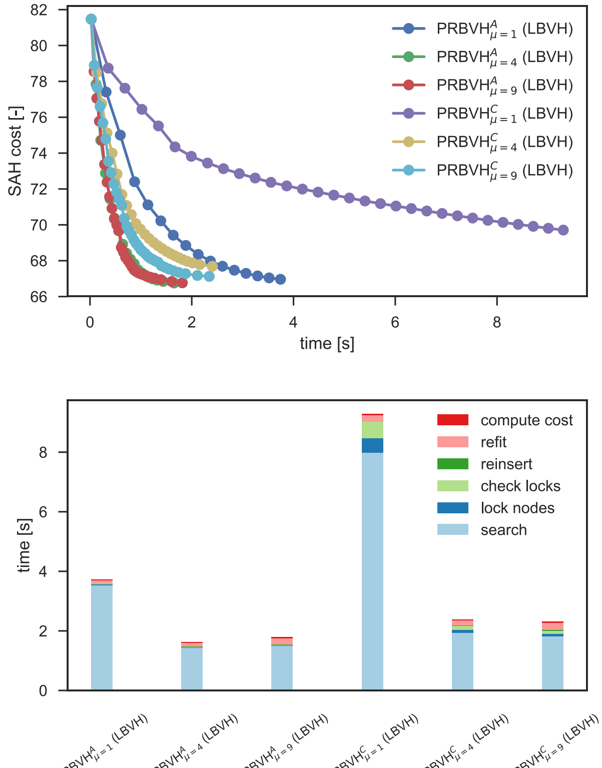

Figure 7: *Comparison between the conservative and aggressive strategies for the Crown scene using BVHs built by LBVH: the SAH cost reduction in time (top) and kernel times of different phases of the optimization (bottom).*

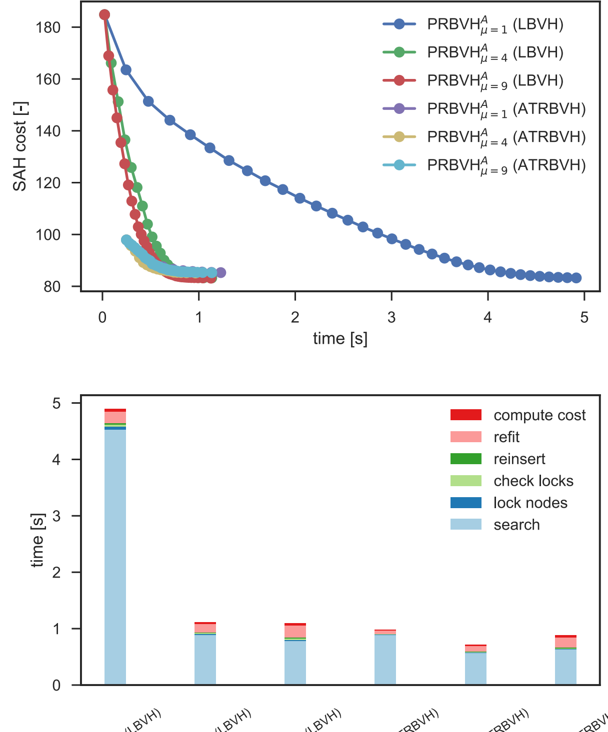

Figure 8: *Comparison between optimization of BVHs initially built by LBVH and ATRBVH for the Manuscript scene using the aggressive strategy: the SAH cost reduction in time (top) and kernel times of different phases of the optimization (bottom).*

[AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High Performance Graphics* (2009), pp. 145–149. 8

[Ape14] APETREI C.: Fast and Simple Agglomerative LBVH Construction. In *Computer Graphics and Visual Computing (CGVC)* (2014). 2

[BHH13] BITTNER J., HAPALA M., HAVRAN V.: Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum 32*, 1 (2013), 85–100. 2, 8, 9

[BWWA17] BENTHIN C., WOOP S., WALD I., AFRA A.: Improved Two-Level BVHs using Partial Re-Braiding. In *Proceedings of High Performance Graphics* (2017). 2

[DP15] DOMINGUES L. R., PEDRINI H.: Bounding Volume Hierarchy Optimization through Agglomerative Treelet Restructuring. In *Proceedings of High-Performance Graphics* (2015), pp. 13–20. 2, 8, 9

[GBDAM15] GANESTAM P., BARRINGER R., DOGGETT M., AKENINE-MÖLLER T.: Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees. *Journal of Computer Graphics Techniques (JCGT) 4*, 3 (2015), 23–42. 2

[GHFB13] GU Y., HE Y., FATAHALIAN K., BLELLOCH G.: Efficient BVH Construction via Approximate Agglomerative Clustering. In *Proceedings of High-Performance Graphics* (2013), pp. 81–88. 2

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and Faster HLBVH with Work Queues. In *Proceedings of High Performance Graphics* (2011), pp. 59–64. 2

[GS87] GOLDSMITH J., SALMON J.: Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Comput. Graph. Appl. 7*, 5 (1987), 14–20. 2

[HHS06] HAVRAN V., HERZOG R., SEIDEL H.-P.: On the Fast Construction of Spatial Data Structures for Ray Tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (2006), pp. 71–80. 2

[HMB17] HENDRICH J., MEISTER D., BITTNER J.: Parallel BVH Construction using Progressive Hierarchical Refinement. *Computer Graphics Forum (Proceedings of Eurographics 2017)* (2017). 2

[HMF07] HUNT W., MARK W. R., FUSSELL D.: Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing* (2007), pp. 47–54. 2

[IWP07] IZE T., WALD I., PARKER S. G.: 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. 2

[KA13] KARRAS T., AILA T.: Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In *Proceedings of High Performance Graphics* (2013), ACM, pp. 89–100. 1, 2

[Kar12] KARRAS T.: Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *Proceedings of High Performance Graphics* (2012), pp. 33–37. 2, 7, 8, 9

[Ken08] KENSLER A.: Tree Rotations for Improving Bounding Volume

Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing* (2008), pp. 73–76. 2

[KK86] KAY T. L., KAJIYA J. T.: Ray Tracing Complex Scenes. *SIGGRAPH Comput. Graph. 20*, 4 (1986), 269–278. 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Comput. Graph. Forum 28*, 2 (2009), 375–384. 2

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer 6*, 3 (1990), 153–65. 1, 2

[MB17] MEISTER D., BITTNER J.: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction . *IEEE Transactions on Visualization & Computer Graphics* (2017). 2, 8, 9

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

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable Parallel Programming with CUDA. *Queue 6*, 2 (2008), 40–53. 7

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. In *Proceedings of High Performance Graphics* (2010), pp. 87–95. 2

[RW80] RUBIN S. M., WHITTED T.: A 3-dimensional Representation for Fast Rendering of Complex Scenes. *SIGGRAPH Comput. Graph. 14*, 3 (1980), 110–116. 2

[VBH17] VINKLER M., BITTNER J., HAVRAN V.: Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction. In *Proceedings of High Performance Graphics* (2017). 2

[Wal07] WALD I.: On Fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing* (2007), pp. 33–40. 2

[WBKP08] WALTER B., BALA K., KULKARNI M., PINGALI K.: Fast Agglomerative Clustering for Rendering. In *IEEE Symposium on Interactive Ray Tracing* (2008), pp. 81–86. 2

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. *ACM Trans. Graph. 26*, 1 (2007). 2

[YKL17] YLITIE H., KARRAS T., LAINE S.: Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In *Proceedings of High Performance Graphics* (2017), pp. 4:1–4:13. 2

Algorithm 1 Pseudocode for searching for the best position for the reinsertion.

1: $\mathbf{b}_{in} \leftarrow \text{BOX}(in)$
2: $\mathbf{b}_{parent} \leftarrow \text{BOX}(\text{PARENT}(in))$
3: $\mathbf{b}_{pivot} \leftarrow EMPTY_BOX$
4: $d_{bound} \leftarrow \text{AREA}(\mathbf{b}_{parent}) - \text{AREA}(\mathbf{b}_{in})$
5: $d_{best} \leftarrow d \leftarrow 0$
6: $out_{best} \leftarrow out \leftarrow \text{SIBLING}(in)$
7: $pivot \leftarrow \text{PARENT}(in)$
8: $down \leftarrow TRUE$
9: **while** $TRUE$ **do**
10: $\quad \mathbf{b}_{out} \leftarrow \text{BOX}(out)$
11: $\quad \mathbf{b}_{merged} \leftarrow \text{UNION}(\mathbf{b}_{in}, \mathbf{b}_{out})$
12: $\quad$ **if** $down$ **then**
13: $\quad\quad d_{direct} = \text{AREA}(\mathbf{b}_{parent}) - \text{AREA}(\mathbf{b}_{merged})$
14: $\quad\quad$ **if** $d_{best} < d_{direct} + d$ **then**
15: $\quad\quad\quad d_{best} \leftarrow d_{direct} + d$
16: $\quad\quad\quad out_{best} \leftarrow out$
17: $\quad\quad$ **end if**
18: $\quad\quad d \leftarrow d + \text{AREA}(\mathbf{b}_{out}) - \text{AREA}(\mathbf{b}_{merged})$
19: $\quad\quad$ **if** $\text{LEAF}(out) \lor d_{bound} + d \leq d_{best}$ **then**
20: $\quad\quad\quad down \leftarrow FALSE$
21: $\quad\quad$ **else**
22: $\quad\quad\quad out \leftarrow \text{LEFT}(out)$
23: $\quad\quad$ **end if**
24: $\quad$ **else**
25: $\quad\quad d \leftarrow d - \text{AREA}(\mathbf{b}_{out}) + \text{AREA}(\mathbf{b}_{merged})$
26: $\quad\quad$ **if** $pivot = \text{PARENT}(out)$ **then**
27: $\quad\quad\quad \mathbf{b}_{pivot} \leftarrow \text{UNION}(\mathbf{b}_{out}, \mathbf{b}_{pivot})$
28: $\quad\quad\quad out \leftarrow \text{PARENT}(out)$
29: $\quad\quad\quad \mathbf{b}_{out} \leftarrow \text{BOX}(out)$
30: $\quad\quad\quad$ **if** $out \neq \text{PARENT}(in)$ **then**
31: $\quad\quad\quad\quad \mathbf{b}_{merged} \leftarrow \text{UNION}(\mathbf{b}_{in}, \mathbf{b}_{pivot})$
32: $\quad\quad\quad\quad d_{direct} \leftarrow \text{AREA}(\mathbf{b}_{parent}) - \text{AREA}(\mathbf{b}_{merged})$
33: $\quad\quad\quad\quad$ **if** $d_{best} < d_{direct} + d$ **then**
34: $\quad\quad\quad\quad\quad d_{best} \leftarrow d_{direct} + d$
35: $\quad\quad\quad\quad\quad out_{best} \leftarrow out$
36: $\quad\quad\quad\quad$ **end if**
37: $\quad\quad\quad\quad d \leftarrow d + \text{AREA}(\mathbf{b}_{out}) - \text{AREA}(\mathbf{b}_{pivot})$
38: $\quad\quad\quad$ **end if**
39: $\quad\quad\quad$ **if** $out = root$ **then**
40: $\quad\quad\quad\quad$ **break**
41: $\quad\quad\quad$ **end if**
42: $\quad\quad\quad out \leftarrow \text{SIBLING}(pivot)$
43: $\quad\quad\quad pivot \leftarrow \text{PARENT}(out)$
44: $\quad\quad\quad down \leftarrow TRUE$
45: $\quad\quad$ **else**
46: $\quad\quad\quad$ **if** $out = \text{LEFT}(\text{PARENT}(out))$ **then**
47: $\quad\quad\quad\quad down \leftarrow TRUE$
48: $\quad\quad\quad\quad out \leftarrow \text{SIBLING}(out)$
49: $\quad\quad\quad$ **else**
50: $\quad\quad\quad\quad out \leftarrow \text{PARENT}(out)$
51: $\quad\quad\quad$ **end if**
52: $\quad\quad$ **end if**
53: $\quad$ **end if**
54: **end while**

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.