Volume xx (200y), Number z, pp. 1–12

Fast Insertion-Based Optimization of Bounding Volume Hierarchies

Jiří Bittner, Michal Hapala and Vlastimil Havran

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

Abstract

We present an algorithm for fast optimization of bounding volume hierarchies (BVH) for efficient ray tracing. We perform selective updates of the hierarchy driven by the cost model derived from the surface area heuristic. In each step the algorithm updates a fraction of the hierarchy nodes in order to minimize the overall hierarchy cost. The updates are realized by simple operations on the tree nodes: removal, search, and insertion. Our method can quickly reduce the cost of the hierarchy constructed by the traditional techniques such as the surface area heuristic. We evaluate the properties of the proposed method on fourteen test scenes of different complexity including individual objects and architectural scenes. The results show that our method can improve a BVH initially constructed with the surface area heuristic by up to 27% and a BVH constructed with the spatial median split by up to 88%.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—[Ray tracing]

1. Introduction

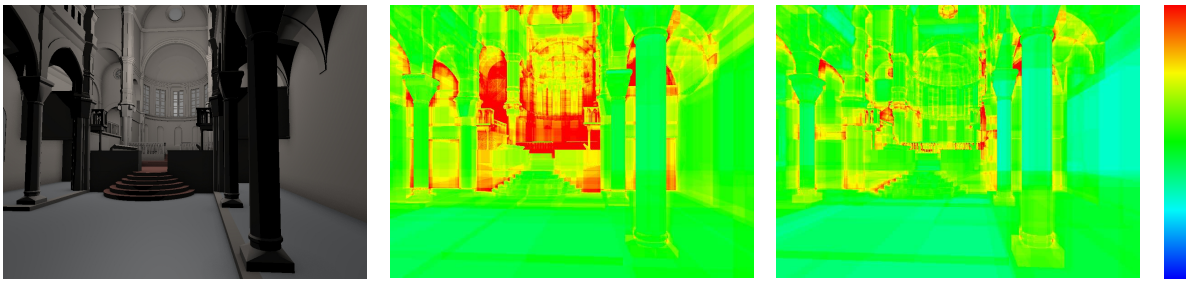
The current ray tracing based algorithms allow to capture even complex illumination effects and reach high degree of realism of the rendered images. With advances in computational power and algorithmic efficiency the ray tracing based methods have also become an alternative to rasterization for interactive and real time applications. Unlike rasterization, ray tracing relies on a highly efficient acceleration data structure which allows to trace tens or hundreds million rays per second.

Constructing such high quality acceleration data structures is thus very important and it has received a strong attention in the last two decades. Even a relatively small improvement in performance can bring an interactive ray tracer closer to real-time or provide significant time savings when rendering many high quality images of complex and detailed scenes in the movie industry.

There are two major classes of data structures used for ray tracing acceleration: spatial subdivisions (such as kd-trees, octrees, or grids) and bounding volume hierarchies (BVH). In this paper we propose a method which allow to optimize a given BVH beyond the current state of the art techniques. The method is based on an iterative algorithm that changes the topology of inner nodes of a bounding volume hierarchy in order to improve the quality of the hierarchy initially built with arbitrary method. An example visualization of the reduction of traversal steps achieved by the proposed method is shown in Figure 1.

Our method constructs more efficient bounding volume hierarchies in shorter time than previous approaches, while it allows to easily trade-off the time used for updating the BVH and the expected traversal cost. Compared to the current state of the art approach for high quality BVHs proposed by Kensler [Ken08] (simulated annealing), our algorithm is 25 to 147 times faster for architectural scenes, while achieving BVHs of comparable or even better quality. The method is applicable to a BVH built with an arbitrary technique, it is simple to implement and thus it has a potential to become a common optimization approach following the construction of the hierarchy.

The paper is further structured as follows. In Section 2 we describe the related work, Section 3 describes the proposed algorithm, Section 4 shows the results and Section 5 concludes the paper.



(a) (b) (c) (d)

Figure 1: *Visualization of the number of traversal steps for the view of the Sibenik Cathedral scene rendered with primary and ambient occlusion rays. (a) Rendered image. (b) Number of traversal steps per ray for BVH built with SAH. (c) Number of traversal steps for the BVH optimized by the proposed method. On this image our method gives an approximately 16% performance gain over the base build algorithm with the SAH. (d) Bar-graph with pseudo colors used in the two visualization images - blue color corresponds to 0 traversal steps, red color to 100 or more steps.*

2. Related Work

There is a large body of literature on efficient spatial data structures for rendering. We restrict our discussion to the BVH methods for efficient rendering of static scenes.

The bounding volume hierarchies have a long tradition in the context of ray tracing. Rubin and Whitted used rectangular bounding volumes to create a BVH manually [RW80]. Weghorst et al. [WHG84] studied different types of bounding volumes for BVH and used a modeling hierarchy to build it. Kay and Kajiya [KK86] suggested to create a BVH by an automatic top down recursive algorithm with a spatial median method and studied the efficiency of different bounding volumes. Goldsmith and Salmon [GS87] proposed the measure currently known as the *surface area heuristic* (SAH) which predicts the efficiency of the hierarchy already during the BVH construction. However, as they have built up the BVH incrementally by insertion, the BVH was usually inefficient as shown in performance study by Havran [Hav00] and later by Masso et al. [ML03]. The improvements for the efficiency of BVH were discussed in several articles in Ray Tracing News [RTN] and suggested in literature more than a decade ago, for example in the paper by Smits [Smi98].

BVHs are mostly used with axis-aligned bounding boxes as bounding volumes. Wächter and Keller [WK06] and Woop et al. [WMS06] used different bounding primitives resulting in light-weight hierarchies. Another research effort was devoted to decreasing the memory size by hierarchical encoding of the bounding volumes [MW06, KMKY10] or their better memory layout with respect to data traffic [YM06].

The precise evaluation of surface area heuristic requires sorting and thus exhibits $O(NlogN)$ complexity (N is the number of scene triangles). To reduce the constants behind the asymptotic complexity Havran et al. [HHS06], Wald et al. [Wal07, WBS07, WIP08], and Ize et al. [IWP07] used approximate SAH evaluation based on binning and centroids of bounding boxes of triangles. Another method to reduce the sorting demands was proposed by Hunt et al. [HMF07] who suggest reusing the organization of geometric data in the scene graph. They show that when certain assumptions about the scene graph hold, it is possible to achieve linear time complexity of BVH construction. Dammertz et al. [DHK08] proposed the variation of BVH using the branching factor of four to gain better utilization of SIMD units in modern CPUs.

Wald [Wal07] proposed a CPU based parallel BVH construction by separating the build up process into two stages — horizontal and vertical parallel execution. Another asynchronous construction method was presented by Ize et al. [IWP07]. More recently, the parallel build-up of BVH has been demonstrated also on GPU by Lauterbach et al. [LGS*09], using 3D space-filling curve. Aila and Laine [AL09] identify the sources of inefficiency for traversing BVH on a GPU and propose a simple solution to improve the performance. Wald studied the possibility of fast rebuilds from scratch on upcoming Intel architecture with many cores [Wal12]. Pantaleoni and Luebke [PL10] and Garanzha et al. [GPM11] recently proposed GPU based methods for parallel BVH construction that are able to very quickly construct a BVH, however for complex scenes the BVH quality for these methods can be significantly lower than that of the full SAH builder.

Several papers have been published that relax the condition of having only one reference to each primitive. Ernst and Greiner [EG07] suggest to precompute several bounding boxes for each primitive and use them independently during top down construction. More recently, to achieve higher performance Stich et al. [SFD09] and Popov et al. [PGDS09] suggest to use spatial splits during the BVH construction, hence reintroducing some properties of kd-trees to BVHs.

Recently more interest has been devoted to methods, which are not limited to top-down BVH construction.

Walter et al. [WBKP08], propose to use bottom-up agglomerative clustering for constructing high quality BVH. Kensler [Ken08] propose to optimize the BVH in postprocess using tree rotations with the hill climbing or simulated annealing optimization algorithms. Both these approaches allow to decrease the expected cost of BVH compared to the top down approach.

The goal of our work is the most similar to the paper of Kensler [Ken08], where the BVH is optimized beyond the common golden standard, i.e. BVHs built in top down fashion with the surface area heuristic. Compared to the method of Kensler [Ken08] our optimization procedure is significantly faster, and for complex scenes it results in hierarchies with lower costs.

3. Selective BVH Updates

This section describes our method starting with the algorithm outline, followed by a detailed description of individual steps of the algorithm, discussion of the design choices and providing further implementation details.

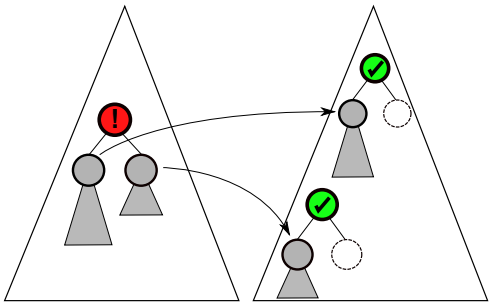


Figure 2: . *The core of our method is the removal of inefficient nodes from the tree and re-insertion of their children to positions that decrease the overall cost of the tree.*

3.1. Algorithm Overview

The BVH as an input of our algorithm can be built with various ways. We construct a BVH using a standard top down technique with the cost model based on the surface area heuristic as used for example in [Wal07, HSZ*11]. Alternatively we can use a faster BVH construction method based on object, or spatial median splits, or the method by [PL10]. Our algorithm performs the following steps (see also Figure 2):

Begin **While Loop**

1) Select inner nodes for optimization
2) For each selected node
   a) Remove both its children from the tree
   b) Find a position to reinsert the children using a cost driven branch and bound search
   c) Insert each of the two children at their new positions and refit bounding volumes of all affected nodes

End **While Loop** (until termination criteria are met)

The core of our method lies in steps 1) and 2) of the algorithm. In step 1) we select the inner nodes for optimization and in step 2) these nodes are removed from the tree and then reinserted back into the tree at more appropriate positions. In the next section we recall the cost model behind our approach and then discuss individual steps of the algorithm in more detail.

3.2. Cost model

The surface area heuristic (SAH) [MB90, Hav00] is usually described using a formula evaluating for each node the expected number of operations for processing a given ray, i.e., the cost of the node. In particular given a node N and assuming uniformly distributed unoccluded rays, which intersect the bounding volume of the node N, the expected cost of the node $C(N)$ is given as:

$$C(N) = \tag{1}$$
$$= \begin{cases} c_T + \frac{SA(L(N))\cdot C(L(N)) + SA(R(N))\cdot C(R(N))}{SA(N)} & \text{if } N \text{ is inner} \\ c_I \cdot t_N & \text{if } N \text{ is leaf} \end{cases},$$

where c_T is the cost of traversing the inner node of the tree including the box intersection calculations, c_I is the cost for ray triangle intersection, t_N is the number of triangles in leaf N, $SA(x)$ is the surface area of the bounding box associated with the node x, and $L(N)$ and $R(N)$ are the left and right children of N, respectively. Note that for hierarchies with larger branching factor than two, the traversal cost c_T would increase.

The SAH makes two assumptions: (1) the distribution of rays is uniform, (2) the rays are unoccluded thus the traversal does not terminate when a ray intersects a geometric primitive. Although these assumptions are generally not met in practice, the experiments indicate that the cost model with the SAH expresses the runtime behavior of a ray tracing quite well (see Figure 3). Therefore reducing the cost is directly reflected in reducing ray tracing times and as we show in Section 4.

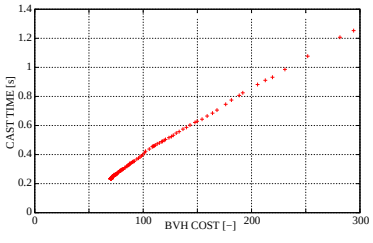


Figure 3: *Dependence of the ray casting time on the cost of the tree (measured on the Sibenik Cathedral scene) for different BVHs. The trees of different costs were obtained during the cost optimization algorithm described in the paper.*

The cost of the root node expresses the expected number of operations to process a ray intersecting the scene. By a simple derivation the recursion can be eliminated and the cost of the tree $C(T)$ can be rewritten as:

$$C(T) = \frac{1}{SA(T)} \left[c_T \cdot \sum_{N \in inner\ nodes} SA(N) + c_I \cdot \sum_{N \in leaves} SA(N) \cdot t_N \right], \quad (2)$$

where $SA(T)$ is the surface area of the bounding box of the scene. Note that the second term in the formula representing the ray triangle intersection calculations is constant for a given scene supposed there is a fixed number of primitives per leaf. Thus the cost term which should primarily be optimized is the sum of surface areas of inner nodes in the tree which induces the traversal overhead of the interior part of the tree ($c_T \cdot \sum SA(N)$). This is exactly the core of our approach - we perform *global updates* of the tree by removing and reinserting nodes to minimize the total sum of surface areas of inner nodes. This contrasts to previous BVH optimization techniques which use local operations on the tree such as rotations.

3.3. Updating Nodes

Let us assume that we have identified a node N in the tree which causes a cost overhead. The key idea of our method is to remove the child nodes of N from the tree and reinsert both of them back at more appropriate positions. For the two child nodes we perform a global search to find the insertion positions that will minimize the tree cost. Once the insertion position is found, the node is inserted in the tree using local operations. Note that although we use a global search for the best position to reinsert the nodes, the method performs a *greedy* optimization since we always choose the positions which minimize the current tree cost.

The rest of this section describes the steps of removing, searching, and reinsertion in more detail. In Section 3.4 we then describe how to actually select the nodes to be updated.

Removing nodes. When updating an inner node N, we remove N, its children L and R, and its parent P from the tree. Then we update the links of the affected nodes to keep the topological consistency of the tree. We also update the bounding boxes of the affected nodes by traversing up to the root of the tree. We put the children L and R in an ordered list of nodes to be reinserted, while the nodes are ordered so that the node with larger surface area will be processed first. The nodes N and P are placed in a list of nodes which will be used to link the reinserted nodes with the nodes at the new positions of the tree. The removal operation is illustrated in Figure 4. Note that nodes L and R need not be leaves, but they can represent whole subtrees of the BVH.

Searching for new positions. We start the search for the new position to insert the node L at the root of the BVH and

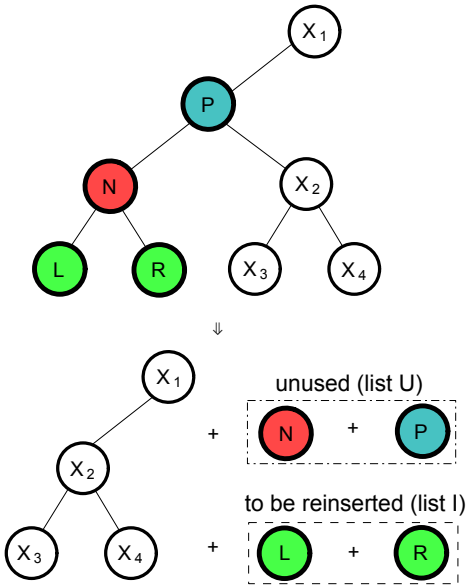


Figure 4: *Illustration of removal of node N – RemoveNode(node N, list U, list I, root P) operation. The children of N are put into the list I that contains the nodes to be inserted. Nodes N and P are put to the list U that stores the nodes that can be reused.*

incrementally compute the total increase of the surface area. When reaching a node X, the surface area and therefore the cost increase is given by two components:

- the *direct cost* $C_D(L,X) = SA(X \cup L)$, where $SA(X \cup L)$ denotes the surface area of the box that is a union of bounding boxes of the node L and the node X.
- the *induced cost* $C_I(L,X)$, that is the accumulated increase of the surface area on the path from the root to the parent of the node X assuming the node L would be inserted in the subtree rooted at X. This can be defined also recursively so $C_I(L,X) = 0$ if X is the root node and $C_I(L,X) = C_I(L, parent(X)) + SA(parent(X) \cup L) - SA(parent(X))$, otherwise.

The two components of the cost increase are illustrated in Figure 5. The total increase of the surface area of the tree is considered as the cost for merging L and X, that is $C(L,X) = C_D(L,X) + C_I(L,X)$. We search for such a node X_{best} that minimizes this cost in the whole tree.

We use a *branch and bound* algorithm based on a priority queue in which the priority is inversely proportional to the induced cost. We can prune the search along the tree effectively using the smallest cost C_{best} corresponding to node X_{best} found so far. We evaluate a lower bound of the cost in the subtree of X in order to decide whether to continue the search in that subtree. The lower bound of the cost is given by the induced cost above X and the surface area of L, which is the lower bound of the direct cost in the whole subtree of X — the induced cost represents the necessary enlargement

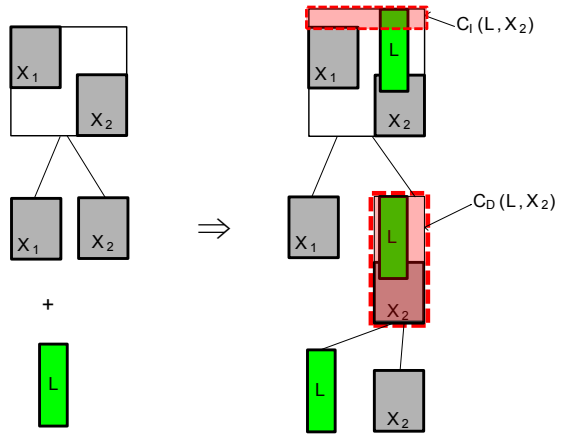


Figure 5: *A 2D example of how the total cost increase of adding the node L to the tree at node X_2 is computed. The induced cost $C_I(L,X_2)$ results from enlarging ascendants of X_2 when inserting the node L as the bounding boxes have to be refitted (Algorithm 1:line 20). The direct cost $C_D(L,X_2)$ is surface area of the box for the union of X_2 and L (Algorithm 1:line 11).*

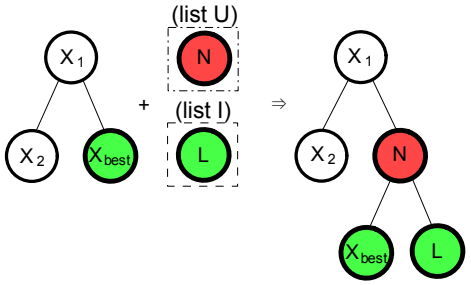


Figure 6: *Illustration of reinserting the node L back to the tree. The node L is merged with the node X_{best} while using the node N as their new parent. The same procedure is used for inserting the node R (using P as the parent node).*

of nodes above X and the surface area of L is the minimum size of the node inserted into the tree, which joins L with a node from the tree. The subtree of X is traversed only if the lower bound of the cost is smaller than C_{best}. The whole algorithm can be terminated if the lower bound of the cost for the node on the top of the priority queue is larger than C_{best}. The pseudocode of the searching algorithm is shown in Algorithm 1.

Note that we have also experimented with using the total cost of the node X for driving the priority queue (instead of the induced cost), but in that case the pruning of the search was not as efficient as for using only the induced cost for computing the priority.

Similar cost model was used in the early work of Goldsmith and Salmon [GS87] who aimed to optimized the tree construction by minimizing the overall cost of the tree during incremental insertion of primitives. They proposed to track the "inheritance cost" which corresponds to our induced cost. However, the actual search of the tree was limited either to a greedy decision and a traversal of a single path or spreading the search in all subtrees for higher levels of the tree. This method was later improved in independent work by Omohundro [Omo89] who proposed to use priority queue when constructing an optimized sphere tree for proximity searching. Our method differs from the search technique of Omohundro by using different metric for priority (induced cost instead of total cost). Also our target application is different as we use the search within the tree update procedure and work with complete subtrees instead of incremental insertion of scene primitives.

```
1  Algorithm:FindNodeForReinsertion(node L)
2  // C_best - the smallest total cost increase found so far
   C_best = infinity;
3  // Priority queue contains pairs: (node, induced cost)
   Push (Root of BVH, 0, 1/ε) to priority queue PQ;
4  while PQ is not empty do
5      (X, C_I(L,X)) = Pop node from PQ;
6      if C_I(L,X) + SA(L) >= C_best then
7          // Early termination - not possible
8          break; // to reduce the cost C_best
9      end
10     // Compute the total cost of merging L with X
11     C_D(L,X) = SA(X ∪ L); // Direct cost
12     C(L,X) = C_I(L,X) + C_D(L,X); // Total cost
13     if C(L,X) < C_best then
14         // Merging L and X decreases the best cost
15         C_best = C(L,X);
16         // X_best is the currently best node found
17         X_best = X;
18     end
19     // Calculate the induced cost for children of X
20     C_I = C(L,X) - SA(X);
21     // Check if the cost decrease is possible in subtree
22     if C_I + SA(L) < C_best then
23         if X is not a leaf then
24             // Search in both children
25             Push (left child of X, C_I, 1/(C_I+ε)) to PQ ;
26             Push (right child of X, C_I, 1/(C_I+ε)) to PQ ;
27         end
28     end
29 end
30 return X_best;
```

Algorithm 1: FindNodeForReinsertion(*node L*) - pseudocode of finding suitable inner node or leaf for reinsertion of the candidate node L so the total cost increase is minimized. Constant ε is a small positive number (e.g. 10^{-20}), the entries with the highest priorities are removed first from the priority queue PQ.

Reinserting nodes. After we select the node X_{best} for insertion, which minimizes the cost increase in the whole BVH, we simply merge X_{best} and L using one of the removed nodes (N or P) as their parent. After each reinsertion we update all the bounding boxes along the path from the parent of the merged nodes to the root. The algorithm is illustrated in Figure 6.

3.4. Selecting Nodes For Update

The update procedure described above process arbitrarily selected nodes in the tree. An obvious choice for the selecting the nodes for updates is random sampling. When using a random sampling we observe that the BVH cost is reduced until a point where it converges.

In order to accelerate the tree optimization we should first update those nodes that cause the highest cost overhead (surface area increase) in the tree. To achieve this we need a node inefficiency measure that would ideally correlate with the actual cost reduction when updating the node. We experimented with numerous node inefficiency measures and below we discuss the three most important ones which we finally combine together.

The first node inefficiency measure M_{SUM} corresponds to a component of the cost model used by Lauterbach et al. [LYTM06] and is evaluated as:

$$M_{SUM}(N) = \frac{SA(N)}{1/|children\ of\ N| \cdot \sum_{X \in children\ of\ N} SA(X)},$$

where $SA(N)$ is the surface area of the evaluated node N, $|children\ of\ N|$ is the number child nodes of N, and $\sum_{X \in children\ of\ N} SA(X)$ is the sum of surface areas for these child nodes. This measure estimates the relative increase of the surface area of the node with respect to average surface areas of the children. Thus if there will be a lot of empty space inside the node this measure will be large.

The second node inefficiency measure M_{MIN} is evaluated as:

$$M_{MIN}(N) = \frac{SA(N)}{Min_{X \in children\ of\ N} SA(X)},$$

where $SA(N)$ is the surface area of the evaluated node and $Min_{X \in children\ of\ N} SA(X)$ is the minimum of surface areas of its children. This measure aims to handle the situation when the node contains child nodes of significantly different sizes (e.g. one large node representing the whole terrain and a small node representing a particular object on the terrain). Then the M_{SUM} measure defined above might not identify such node as problematic as it takes the average surface area of the children which in this example will still be large. On the contrary the M_{MIN} measure will detect such a situation.

The third node inefficiency measure M_{AREA} directly corresponds to the surface area of the node:

$$M_{AREA}(N) = SA(N).$$

The M_{AREA} measure simple prioritizes the updates of the larger nodes of the tree since each such node has a significant contribution to the tree cost.

These three measures are able to detect some situations in which high global cost decrease can be expected. We have experimentally verified that the steepest decrease of the tree cost as a function of computation time was consistently achieved by combining these three measures together into:

$$M_{COMB}(N) = M_{SUM}(N) \cdot M_{MIN}(N) \cdot M_{AREA}(N)$$

Once the node inefficiency measure is defined we can use it to prioritize the updates of the hierarchy nodes according to their inefficiency measure. Our method works in passes where in each pass it updates a specified number of nodes k (typically k=1% of nodes). These nodes are selected as follows: we evaluate the inefficiency measure for all inner nodes. Then we determine k nodes with the highest values of the inefficiency measure using a partial sort of the node array. These k nodes are then processed sequentially in descending order according to their inefficiency measures (the most inefficient nodes first).

Note that an alternative would be to always update a single node with currently the highest inefficiency measure. The batch processing of nodes however speeds up the node selection procedure since the inefficiency measures are calculated only once per pass and are not updated after each change in the tree. Additionally the batch processing makes the method more robust with respect to getting stuck in a local minimum for the case that the inefficiency measure of some node(s) is hard to reduce. A comparison of the decrease of the cost using the above described cost measures is shown in Figure 7.

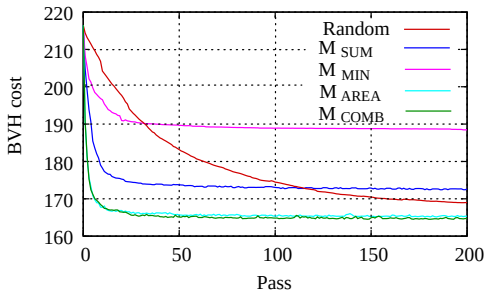


Figure 7: *Cost reduction using a random selection and the proposed node inefficiency measures on the Soda Hall scene. Note that the area measure and the combined measure achieve very fast decrease of the expected cost from 216 to 165. In one pass we update $k = 1\%$ of nodes.*

3.5. Terminating the BVH optimization

In the beginning of the optimization the vast majority of updates lead to reduction of the BVH cost. Since the removal operation removes two nodes from the tree and processes

them sequentially (the first child is inserted in the tree while the second child is still removed), however, it is possible that after reinserting both nodes to the tree the BVH cost will increase. This behavior becomes more apparent when the optimization converges and the BVH cost cannot be reduced anymore. Then the BVH cost oscillates in a small range near the reached minimum. Note that by a simple modification of the method which would always remove just one child from the tree we could ensure that the cost is either reduced in the given step or it remains the same. However, our experiments have shown that the BVH cost is reduced slightly more if we use the method of removing both children, although temporarily the optimization step might provide a small cost increase.

As the optimization is progressively reducing the cost we can use different termination criteria deciding when to stop the optimization such as the maximum time or the number of passes. The criteria can also be based on evaluating the convergence of the cost. We propose to terminate the computation when the cost does not improve within a given number of update passes p_T (recall that each pass updates certain number of nodes).

When using the combined inefficiency measure the cost might stabilize at a slightly higher value than when using random sampling as there are some nodes that are never selected for optimization since their measure is low. To avoid this behavior we switch to random sampling of nodes when we detect that the cost reduction becomes very low or even zero. Similarly to the termination of the whole computation this decision is made not for a single node but for sequence of processed nodes. We switch to the random selection if in the given number of passes p_R ($p_R \leq p_T$) the cost of the BVH does not reduce.

3.6. BVH Tree Compaction

We assumed that the BVH trees are constructed until each leaf contains a single triangle (or a geometric primitive in general). It is usually more beneficial to construct leaves with more triangles (e.g. 8 to 10) following the actual traversal and intersection constants (c_T and c_I) used in the SAH cost model [ML03, HHS06, Wal07, WBS07].

To get the most benefit from our optimization method we apply the method in two phases. First, we build the tree so that each leaf contains a single triangle. Second, we run the postprocessing phase using a post-order traversal of the whole tree and evaluate the cost of each node using Eq. 1. This requires also counting the number of triangles associated with the node during the post-order traversal. Whenever the cost of an interior node N is larger than the cost for a leaf created for the triangles contained in the leaves of the subtree rooted in the node N, we collapse the subtree to a leaf that references all the corresponding triangles. As a result the cost of the compacted tree can be significantly reduced compared to the tree with a single triangle per leaf (see the ratios between the *cost* and *ocost* in Section 4). Note that this postprocessing phase happens after the optimization and prior to the use of BVH for ray tracing. The time needed for this phase is linear with the number of nodes and presents almost no computation time overhead.

Note that our tree optimization method can be easily applied even to BVHs with more triangles per leaf (i.e. those compacted prior to the optimization). In this case we have a lower number of nodes in the BVH which reduces the optimization time. On the other hand the assignment of triangles to the leaves will not be changed, although we could potentially find a tree with better distribution of triangles to leaves which reduces the tree cost.

3.7. Discussion

The idea of our optimization algorithm is to repeatedly optimize the BVH, which is similar to the paper of Kensler [Ken08]. There are two major differences in the core of the optimization procedure that result in a different behavior of our method. First, Kensler's method *always traverses the whole tree* in a depth-first-search order to change the hierarchy around the traversed nodes. Our approach, on the other hand, selects the nodes to be optimized either randomly or using importance based decision. Second, the Kensler's method *changes the topology of the tree only locally* using one of four possible rotations. Although the sequence of rotations can principally lead to any BVH including the tree with a globally optimized cost, the changes in a BVH for one optimization step are much smaller than in our algorithm, where the subtrees are removed and then reinserted into the most appropriate position. That is, in our algorithm one optimization step can change (and often changes) the tree globally and hence has the potential of improving the cost more efficiently.

Unlike Eisemann et al. [EGMM07] our search for the best candidate to reinsert the node intentionally starts at the root to find the best suitable position in the scope of the whole tree. Using the upwards traversal in the method of Eisemann et al. [EGMM07] reduces the number of traversal steps during the optimization, but does not necessarily find a good position for the inserted node.

4. Results

We have implemented the proposed algorithm in a single threaded C++ application. The results of BVH optimization and CPU ray tracing were evaluated on a PC with Linux OS, Intel(R) Xeon(R) CPU E5440 with the frequency 2.83GHz and 40GB of RAM, and GNU C compiler version 4.40. To compute the cost according to equations 1 and 2 we used $c_T = 3.0$ and $c_I = 2.0$. As termination criteria described in Section 3.4 and 3.5 we used $k = 1\%$, $p_T = 10$, and $p_R = 5$.

We have evaluated the method on fourteen models of different complexity which we classify into two distinct groups: (1) *individual objects* and (2) *architectural scenes*, i.e., models of larger spatial extent typically containing many objects. We have constructed the initial BVH by top down recursive procedure using a precise SAH builder which evaluates all discontinuities (two positions for each triangle) in the cost function for all three axes. The rendered images for objects and scenes are shown in Figure 9 and 10.

4.1. BVH Cost Reduction

We discuss separately the results for BVH built for individual objects and architectural scenes. As a reference for comparison we use our reimplementation of the state of the art BVH optimization algorithms of Kensler [Ken08] for both hill climbing and simulated annealing. For all tested methods we evaluate the BVH cost (denoted as *cost*), BVH cost optimized by compacting the tree with the method described in Section 3.6 (*ocost*), the time for optimizing the tree until predefined termination criteria ($CPUupdate$), the total build time including the optimization ($CPUbuild$) and the times for ray casting on the CPU and on the GPU for different ray types (primary rays, random rays, ambient occlusion rays).

Individual objects. As individual objects we have used five geometric data sets also used by Kensler's paper. These models exhibit relatively uniform distribution of triangles and low depth complexity. Therefore the traditional top down build algorithm with the SAH constructs a high quality BVH that is difficult to optimize. For these scenes the updates provide practically no reduction of the BVH cost (see Table 2). It is worth to mention the 5% cost reduction for the Hairball scene, which results from its higher geometrical complexity (and hence possible source of inefficiency).

For BVHs initially constructed using spatial median splits our method optimizes the BVH cost close to the cost achieved by the top down SAH algorithm. The results are given in the Table 3. The final cost achieved by our algorithm is decreased the same or more than in the reference algorithm, the simulated annealing by Kensler [Ken08], however, the total build time achieves the great speedup, our algorithm is up to 10 to 25 times faster.

Architectural scenes. The results for nine more complex architectural scenes show our method can reduce the initial cost of the tree constructed with the traditional top down algorithm with SAH by 4% to 24% with an average of 17% (see Table 4). This cost reduction was achieved in time which was about three times smaller than the time of the initial tree construction (albeit the implementation of the exact SAH builder is not optimized). We can observe that the cost reduction of the BVH optimized by our method is larger than that of the current state of the art methods for high quality BVH proposed by Kensler [Ken08]. The build time for the hill climbing reference method is slightly lower than for our method (note that this also depends on the particular termination criteria used in our method), but the hill climbing is not able to reduce the cost more than by a few percent.

The simulated annealing of Kensler [Ken08] achieves better cost reduction than the hill climbing, while the running time of the method is about two order of magnitude higher than for hill climbing. Our method however provides BVHs with even lower cost (up to 10% difference) than the simulated annealing by Kensler in computation time from 28 to 80 times smaller. This brings us to an observation that our proposed technique is currently able to construct the best known BVHs for the given scene and the cost model based on SAH. The BVH quality improvement over the previous state-of-the-art method is not dramatic, however the speed in which we obtain these improvements is significant (almost two orders of magnitude compared to the simulated annealing), which can actually lead to using the proposed technique in practice as the BVH build time is only slightly higher than without our method.

For BVHs built top down using a simple spatial median split the cost reduction achieved by our algorithm is very significant. The results are given in Table 5. Interestingly, the BVH cost quickly converges almost to the same cost as for the case when the BVH was built with top down build algorithm with SAH and optimized by our algorithm, but the optimization takes more computation time. This behavior can be seen in Figure 8 where we show the optimization process using 200 passes and when updating 1% of inner nodes per pass. Again, the time needed for the build including optimization is 25 to 147 times smaller than the time of simulated annealing by Kensler.

In order to provide more compact overview of the results we have summarized all results for the architectural scenes in Table 1, where the cost without the reduction described in Section 3.6 is reported. The cost and time for the top down BVH build with SAH is taken there as a reference.

BVH structure analysis. In order to find out what particular changes to the BVH our algorithm does we calculated histograms of the surface areas of the nodes at different depths of the tree for the initial BVH and for the BVH optimized by our method. These histograms are shown in Figure 11.

The plots show that our optimization method reduces the sum of surface areas of inner nodes especially for the middle range depths. We can also observe that this is achieved by restructuring the tree so that certain nodes are placed deeper in the tree.

BVH tree compaction. We also evaluated the contribution of the tree compaction described in Section 3.6. The *ocost* after the compaction can be non-negligibly lower than the initial *cost* before performing the tree compaction. In particular, the *ocost* is by about 8% lower than *cost* for individual objects. For architectural scenes the difference is

even more significant, for BVH top down build with SAH it reaches 20% and for the BVH built with spatial median 17% on average. Interestingly, the impact of our tree optimization algorithm is typically a few percent larger when considering the ratio of *ocost* than the ratio of *cost* before and after the optimization. For example for the Soda Hall scene the *cost* is reduced by 24% (ratio 0.76), while the *ocost* is reduced by 27% (ratio 0.73) (see Table 4).

4.2. Ray Tracing Performance

We have evaluated the optimized BVH using two different ray tracers. The first one is a CPU based ray tracer with no low level optimizations, the second one is a GPU ray tracer derived from the implementation of Karras et al.'s [KAL09]. For the GPU ray tracer we have used an adaptor that converts the main memory data structures to the data structures used by the GPU application's CUDA kernels.

CPU ray tracer results. The CPU ray tracing performance of the constructed BVH has been evaluated for two scenarios – casting primary rays and shooting random rays. The figures computed for casting primary rays are shown in Figure 10. Note that the time for tracing rays is in strong correlation with the cost irrespective of the scenario for both primary and random rays for all reported results.

GPU ray tracer results. The GPU ray tracing results were evaluated using a modified version of the Karras et al.'s [KAL09] GPU ray tracer. For measurements we have used a PC with Intel Core i7-2600K 3.40GHz, NVIDIA GeForce GTX 580 3GB GDDR5 and Windows 7 OS. The application was built using Microsoft Visual Studio 2010, version 64-bit, with CUDA Toolkit v3.2.

We have tested the performance of primary rays, random rays, and ambient occlusion rays. We can see that the GPU results correlate with the results from the CPU raytracer (see Table 2 and Table 4). Karras et al.'s primary rays and ambient occlusion rays generation and general tracing kernels were used without changes. We have implemented a random rays generation routine, casting 8 million rays for each scene, where the rays are defined by generating two uniformly distributed points in the scene bounding box. For the ambient occlusion test we spawn ten rays at each hit of the primary ray.

There are scenes, where the GPU version does not provide a performance gain comparable to the CPU implementation particularly for primary rays (e.g. Conference - CPU primary 69%, GPU primary 98%), though in these cases the reference method of Kensler exhibits similar behavior. On the contrary on a few tested scenes the optimized BVH provides slightly higher performance gain for the GPU ray tracer than for the CPU version (e.g. Sibenik - CPU primary 85%, GPU primary 81%). The analysis why this happens is a matter of future work when convenient profiling tools on a GPU will become available.

5. Conclusion and Future Work

We proposed an algorithm for building a high quality BVH by incremental updates of the BVH initially constructed by a top down method with surface area heuristic.

The method is based on performing selective updates of the BVH by identifying problematic nodes and reinserting them back in appropriate positions in order to minimize the total BVH cost. The updates are prioritized and the resulting method is highly flexible in terms of the update time with respect to the quality of the hierarchy.

We have shown that for complex scenes our method achieves very good cost reduction in much shorter time than previous methods. In fact the results indicate that the method constructs the best currently known BVHs under the SAH cost model and thus it has a potential to become a common optimization technique, which further reduces the cost of the SAH builders used in practical applications.

Currently, we work on a parallel version of the algorithm on modern GPUs in CUDA, where both the update and the rendering algorithms run solely on the GPU. We also want to study the properties of the hierarchy for other visibility computations such as occlusion and view-frustum culling for large scenes. Another possible future work is to study other types of bounding volume hierarchies such as those with explicit spatial splits or with higher branching factor.

Acknowledgements

We would like to thank the contributors of the scenes used in our paper, Prof. C. Sequin for Soda Hall model, the University of North Carolina for the Power Plant model, Marko Dabrovic for the Sponza and Sibenik models, Ingo Wald for Fairy Forest, Greg Ward for the Conference model, Samuli Laine and Tero Karras for Hairball model, and Stanford repository for other models.

Further, we want to thank Andrew Kensler for providing us with the source code for his paper [Ken08] that allowed to reproduce the reference method exactly. We would also like to thank Tero Karras, Timo Aila, and Samuli Laine for releasing their GPU ray tracing framework. Our research was supported by the Czech Science Foundation under research programs P202/11/1883 (Argie) and P202/12/2413 (Opalis), and the Grant Agency of the Czech Technical University in Prague, grant No. SGS10/289/OHK3/3T/13, supported by Ministry of Education of the Czech Republic.

References

[AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High-Performance Graphics (HPG'09)* (Aug 2009), pp. 145–149. 2

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum 27* (June 2008), 1225–1233(9). 2

Scene	cost[-] SAH build	cost, our optimized SAH build	cost, spatial median	cost, our optimized sp. median	time[s], SAH build	time, our optimized SAH build	time, spatial median	time, our optimized sp. median
Conference	130.30 (100%)	78.66%	646.38%	78.83%	3.45 (100%)	131.30%	12.75%	226.38%
Fairy Forest	95.10 (100%)	96.21%	194.52%	97.70%	1.96 (100%)	120.92%	12.24%	63.27%
Sibenik Cathedral	82.30 (100%)	84.45%	474.74%	86.60%	0.66 (100%)	209.09%	13.64%	175.76%
Sponza	220.20 (100%)	82.74%	571.25%	83.30%	0.56 (100%)	180.36%	12.50%	376.79%
Soda Hall	216.50 (100%)	76.40%	644.96%	76.61%	40.96 (100%)	137.60%	10.47%	187.26%
Power Plant, sec. 9	57.90 (100%)	89.46%	325.87%	90.33%	1.36 (100%)	119.12%	11.76%	205.15%
Power Plant, sec. 16	93.50 (100%)	85.35%	540.16%	85.24%	4.70 (100%)	159.57%	12.77%	378.51%
Power Plant	115.80 (100%)	84.46%	571.10%	85.16%	396.00 (100%)	121.46%	9.08%	71.90%
Pompeii Ten	252.90 (100%)	86.20%	303.24%	86.98%	102.00 (100%)	175.49%	10.76%	114.02%

Table 1: *Summary results for our algorithm used for architectural scenes, where the reference method is SAH-based build method in top down fashion without optimization of the tree (100%). The data are compacted from Tables 4 and 5.*

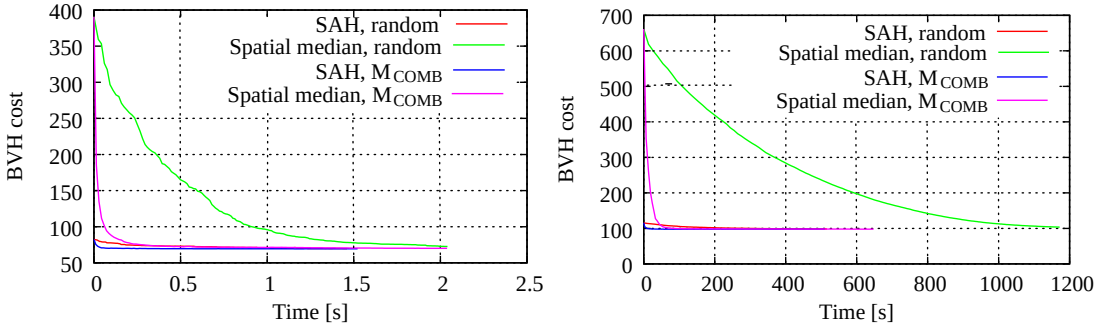


Figure 8: *BVH cost optimization for BVHs built by spatial median and SAH. (left) Sibenik Cathedral, (right) Power Plant. Our method can quickly optimize a tree, which was built either with spatial median or SAH. Note that both trees converge to very similar costs and the convergence is significantly faster for the M_{COMB} metrics (the plots show 200 update passes with 1% of updated nodes per pass).*

[EG07] ERNST M., GREINER G.: Early Split Clipping for Bounding Volume Hierarchies Early Split Clipping for Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing (RT'2007)* (Sept 2007), pp. 73–78. 2

[EGMM07] EISEMANN M., GROSCH T., MAGNOR M., MUELLER S.: Automatic Creation of Object Hierarchies for Ray Tracing Dynamic Scenes. In *WSCG'2007 Short Papers Post-Conference Proceedings* (Jan 2007), Skala V., (Ed.), WSCG, pp. 57–64. 7

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG'11)* (Aug 2011), pp. 59–64. 2

[GS87] GOLDSMITH J., SALMON J.: Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20. 2, 5

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Nov 2000. 2, 3

[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* (Sept 2006), pp. 71–80. 2, 7

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

[HSZ*11] HOU Q., SUN X., ZHOU K., LAUTERBACH C., MANOCHA D.: Memory-Scalable GPU Spatial Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics 17*, 4 (Apr 2011), 466–474. 3

[IWP07] IZE T., WALD I., PARKER S. G.: Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization 2007 (EGPGV'07)* (May 2007), pp. 101–108. 2

[KAL09] KARRAS T., AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs; Google Code, 2009. 9

[Ken08] KENSLER A.: Tree Rotations for Improving Bounding Volume Hierarchies. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing* (Aug 2008), pp. 73–76. 1, 3, 7, 8, 9

[KK86] KAY T. L., KAJIYA J. T.: Ray tracing complex scenes. In *SIGGRAPH '86 Proceedings* (Aug. 1986), Evans D. C., Athay R. J., (Eds.), vol. 20, pp. 269–278. 2

[KMKY10] KIM T.-J., MOON B., KIM D., YOON S.-E.: RACBVHs: Random-Accessible Compressed Bounding Volume

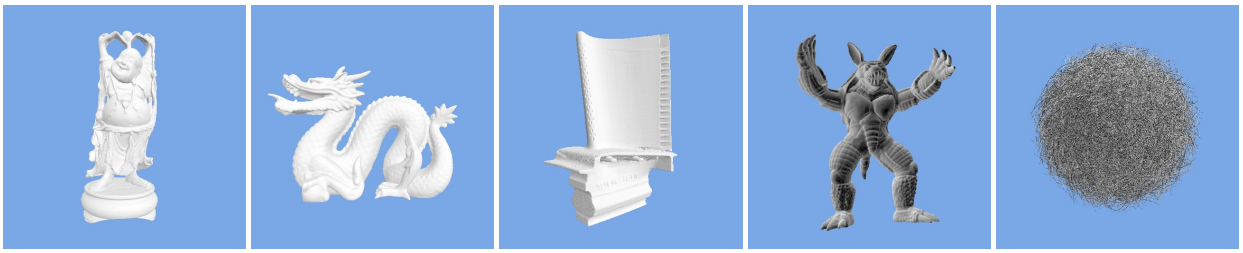


Figure 9: *Snapshots of scenes representing individual objects: Happy Buddha, Dragon, Blade, Armadillo, and Hairball.*

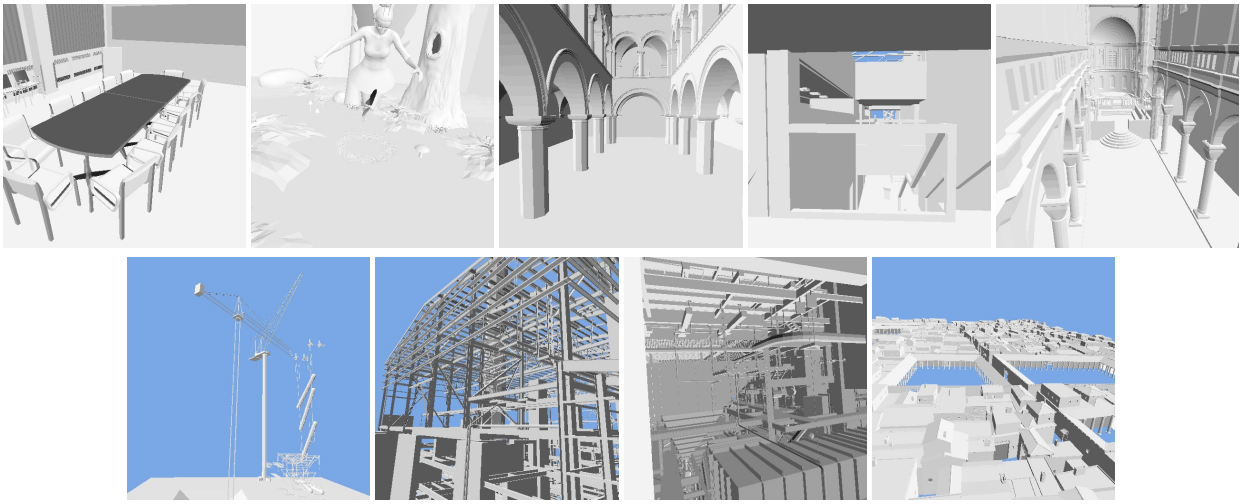


Figure 10: *Snapshots of architectural scenes: Conference, Fairy Forest, Sponza, Sodahall, Sibenik Cathedral, Power Plant section 9, Power Plant section 16, Power Plant, and Pompeii Ten.*

Hierarchies. *IEEE Transactions on Visualization and Computer Graphics 16*, 2 (March-April 2010), 273 –286. 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

[LYTM06] LAUTERBACH C., YOON S.-E., TUFT D., MANOCHA D.: RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *IEEE Symposium on Interactive Ray Tracing (RT'06)* (Sept 2006), pp. 39–46. 6

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6 (1990), 153–65. 3

[ML03] MASSO J. P. M., LOPEZ P. G.: Automatic Hybrid Hierarchy Creation: a Cost-model Based Approach. *Computer Graphics Forum 22*, 1 (2003), 5–13. 2, 7

[MW06] MAHOVSKY J., WYVILL B.: Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. *Computer Graphics Forum 25* (Jun 2006), 173–182(10). 2

[Omo89] OMOHUNDRO S. M.: *Five Balltree Construction Algorithms*. Tech. Rep. TR-89-063, International Computer Science Institute, Berkeley, Nov 1989. 5

[PGDS09] POPOV S., GEORGIEV I., DIMOV R., SLUSALLEK P.: Object partitioning considered harmful: space subdivision for BVHs. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)* (Aug 2009), pp. 15–22. 2

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics (HPG'10)* (Jun 2010), pp. 87–95. 2, 3

[RTN] Ray Tracing News, internet publications, put together and edited by Eric Haines, since 1987. http://tog.acm.org/resources/RTNews/html. 2

[RW80] RUBIN S. M., WHITTED T.: A 3-Dimensional Representation for Fast Rendering of Complex Scenes. In *SIGGRAPH '80 Proceedings* (July 1980), vol. 14, pp. 110–116. 2

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *proceedings of the conference on High Performance Graphics 2009 (HPG'09)* (Aug 2009), pp. 7–13. 2

[Smi98] SMITS B.: Efficiency issues for ray tracing. *Journal of Graphics Tools 3*, 2 (1998), 1–14. 2

[Wal07] WALD I.: On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (Sep 2007), pp. 33–40. 2, 3, 7

[Wal12] WALD I.: Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 1 (Jan 2012), 47–57. 2

[WBKP08] WALTER B., BALA K., KULKARNI M., PINGALI K.: Fast Agglomerative Clustering for Rendering. In *IEEE*

M	cost [-]	ocost [-]	CPU update [s]	CPU build [s]	CPU primary [s]	CPU random [s]	GPU primary [ms]	GPU random [ms]	GPU amb. occlusion [ms]
Happy Buddha (1,087k triangles)									
1	165.3 (1.00)	156.5 (1.00)	0.00	15.78 (1.00)	0.20 (1.00)	2.71 (1.00)	0.94 (1.00)	165.5 (1.00)	6.45 (1.00)
2	161.9 (0.97)	154.2 (0.98)	26.35	42.14 (2.66)	0.20 (1.01)	2.66 (0.98)	0.97 (1.03)	163.3 (0.98)	6.49 (1.00)
3	163.2 (0.98)	155.3 (0.99)	15.04	30.82 (1.95)	0.20 (1.02)	2.72 (1.00)	0.95 (1.01)	163.0 (0.98)	6.48 (1.00)
4	164.2 (0.99)	156.3 (0.99)	2.58	18.37 (1.16)	0.20 (1.00)	2.71 (0.99)	0.94 (1.00)	164.7 (0.99)	6.51 (1.00)
5	165.3 (1.00)	156.5 (1.00)	1135.00	1151.00 (72.92)	0.20 (1.00)	2.71 (1.00)	0.94 (1.00)	165.5 (1.00)	6.45 (1.00)
Dragon (871k triangles)									
1	145.4 (1.00)	138.1 (1.00)	0.00	11.96 (1.00)	0.26 (1.00)	2.37 (1.00)	1.00 (1.00)	156.8 (1.00)	8.80 (1.00)
2	144.5 (0.99)	137.9 (0.99)	6.21	18.18 (1.51)	0.27 (1.03)	2.41 (1.01)	1.07 (1.07)	155.9 (0.99)	8.92 (1.01)
3	145.4 (1.00)	138.1 (1.00)	1.21	13.18 (1.10)	0.26 (1.00)	2.38 (1.00)	1.08 (1.08)	155.6 (0.99)	8.88 (1.00)
4	144.7 (0.99)	138.0 (0.99)	1.80	13.77 (1.15)	0.26 (1.00)	2.37 (0.99)	1.01 (1.01)	156.4 (0.99)	8.85 (1.00)
5	145.4 (1.00)	138.1 (1.00)	911.00	923.00 (77.14)	0.26 (1.00)	2.37 (1.00)	1.00 (1.00)	156.8 (1.00)	8.80 (1.00)
Blade (1.765k triangles)									
1	190.3 (1.00)	178.8 (1.00)	0.00	27.37 (1.00)	0.21 (1.00)	2.35 (1.00)	0.97 (1.00)	10.4 (1.00)	4.01 (1.00)
2	190.3 (1.00)	178.8 (1.00)	3.24	30.61 (1.11)	0.22 (1.00)	2.38 (1.01)	0.97 (1.00)	10.8 (1.03)	4.03 (1.00)
3	190.3 (1.00)	178.8 (1.00)	2.35	29.72 (1.08)	0.22 (1.00)	2.38 (1.01)	0.99 (1.02)	10.7 (1.02)	4.03 (1.00)
4	190.1 (0.99)	178.7 (0.99)	4.65	32.02 (1.16)	0.21 (1.00)	2.34 (0.99)	0.97 (1.00)	10.4 (1.00)	4.00 (0.99)
5	190.3 (1.00)	178.8 (1.00)	1835.00	1863.00 (68.05)	0.21 (1.00)	2.35 (1.00)	0.97 (1.00)	10.4 (1.00)	4.01 (1.00)
Armadillo (307k triangles)									
1	86.3 (1.00)	82.5 (1.00)	0.00	3.35 (1.00)	0.22 (1.00)	1.53 (1.00)	0.99 (1.00)	56.7 (1.00)	6.81 (1.00)
2	86.3 (0.99)	82.5 (1.00)	0.33	3.69 (1.10)	0.22 (0.99)	1.52 (0.99)	0.97 (0.97)	57.7 (1.01)	6.94 (1.01)
3	86.3 (1.00)	82.5 (1.00)	0.34	3.70 (1.10)	0.22 (0.99)	1.52 (0.99)	0.94 (0.94)	57.1 (1.00)	6.91 (1.01)
4	86.2 (0.99)	82.5 (1.00)	0.52	3.87 (1.15)	0.22 (0.99)	1.52 (0.99)	0.97 (0.97)	56.8 (1.00)	6.81 (1.00)
5	86.3 (1.00)	82.5 (1.00)	316.00	319.00 (95.13)	0.22 (1.00)	1.53 (1.00)	0.99 (1.00)	56.7 (1.00)	6.81 (1.00)
Hairball (2,850k triangles)									
1	1415.2 (1.00)	1057.1 (1.00)	0.00	48.75 (1.00)	1.01 (1.00)	7.30 (1.00)	5.55 (1.00)	180.1 (1.00)	43.70 (1.00)
2	1345.3 (0.95)	985.9 (0.93)	143.00	192.00 (3.93)	0.97 (0.96)	6.61 (0.90)	5.09 (0.91)	165.0 (0.91)	40.72 (0.93)
3	1345.7 (0.95)	986.0 (0.93)	131.00	180.00 (3.69)	0.99 (0.98)	6.82 (0.93)	5.01 (0.90)	165.1 (0.91)	40.89 (0.93)
4	1408.0 (0.99)	1052.2 (0.99)	10.09	58.85 (1.20)	1.05 (1.04)	8.99 (1.23)	5.54 (0.99)	179.8 (0.99)	43.70 (1.00)
5	1409.9 (0.99)	1052.8 (0.99)	2948.00	2996.00 (61.57)	1.05 (1.04)	8.76 (1.20)	5.73 (1.03)	185.0 (1.02)	45.07 (1.03)

Table 2: *Results of the BVH optimization for individual objects. The base hierarchy for all methods (M) is created with a SAH top down driven build (method 1). For methods 2 to 5 the initially constructed BVH is followed by an optimization phase using different methods and settings: 2 - our method with random selection, 3 - our method with the combined measure of inefficiency, 4 - Kensler's method, hill climbing, and 5 - Kensler's method, simulated annealing. The values in brackets are the computed ratios against reference values (1.00).*

Symposium on Interactive Ray Tracing (RT'2008) (Aug 2008), pp. 81–86. 3

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (Jan. 2007). 2, 7

[WHG84] WEGHORST H., HOOPER G., GREENBERG D. P.: Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*, 1 (Jan. 1984), 52–69. 2

[WIP08] WALD I., IZE T., PARKER S. G.: Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Comput. Graph. (Special Section: Parallel Graphics and Visualization) 32*, 1 (2008), 3–13. 2

[WK06] WÄCHTER C., KELLER A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Proceedings of the 17th Eurographics Symposium On Rendering (EGSR'06)* (Jun 2006), pp. 139–149. 2

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of conference on Graphics Hardware 2006* (Sep 2006), pp. 67–77. 2

[YM06] YOON S.-E., MANOCHA D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum 25* (Jun 2006), 507–516. 2

M	cost [-]	ocost [-]	CPU update [s]	CPU build [s]	CPU primary [s]	CPU random [s]	GPU primary [ms]	GPU random [ms]	GPU amb. occlusion [ms]
Happy Buddha (1,087k triangles)									
1	275.6 (1.00)	269.1 (1.00)	0.00	1.89 (1.00)	0.34 (1.00)	4.83 (1.00)	1.64 (1.00)	278 (1.00)	12.37 (1.00)
2	166.4 (0.60)	158.7 (0.58)	37.82	39.71 (20.97)	0.21 (0.62)	2.78 (0.57)	0.97 (0.59)	167 (0.60)	6.59 (0.53)
3	168.9 (0.61)	161.4 (0.59)	43.62	45.51 (24.03)	0.21 (0.63)	2.83 (0.58)	1.08 (0.65)	170 (0.61)	6.83 (0.55)
4	217.5 (0.78)	211.4 (0.78)	7.88	9.77 (5.16)	0.27 (0.79)	3.79 (0.78)	1.34 (0.81)	220 (0.79)	9.07 (0.73)
5	179.4 (0.65)	171.8 (0.63)	1152.00	1154.00 (609.33)	0.24 (0.70)	3.14 (0.65)	1.14 (0.69)	181 (0.65)	7.18 (0.58)
Dragon (871k triangles)									
1	232.9 (1.00)	227.6 (1.00)	0.00	1.49 (1.00)	0.46 (1.00)	3.95 (1.00)	2.05 (1.00)	246 (1.00)	16.01 (1.00)
2	146.2 (0.62)	139.6 (0.61)	32.58	34.07 (22.85)	0.29 (0.62)	2.44 (0.61)	1.17 (0.57)	158 (0.63)	9.06 (0.56)
3	149.3 (0.64)	143.0 (0.62)	39.50	40.99 (27.49)	0.29 (0.63)	2.53 (0.64)	1.19 (0.58)	160 (0.65)	9.26 (0.57)
4	189.3 (0.81)	184.2 (0.80)	7.14	8.63 (5.79)	0.38 (0.82)	3.31 (0.83)	1.57 (0.76)	207 (0.83)	12.44 (0.77)
5	157.1 (0.67)	150.5 (0.66)	927.02	928.51 (622.84)	0.33 (0.70)	2.73 (0.69)	1.30 (0.63)	173 (0.70)	10.05 (0.62)
Blade (1.765k triangles)									
1	345.9 (1.00)	337.6 (1.00)	0.00	3.03 (1.00)	0.40 (1.00)	4.53 (1.00)	2.02 (1.00)	21.5 (1.00)	7.95 (1.00)
2	196.3 (0.56)	185.2 (0.54)	82.97	86.00 (28.34)	0.24 (0.60)	2.45 (0.54)	1.10 (0.54)	11.4 (0.53)	4.22 (0.53)
3	202.4 (0.58)	191.5 (0.56)	100.46	103.49 (34.10)	0.25 (0.61)	2.52 (0.55)	1.22 (0.60)	11.8 (0.54)	4.41 (0.55)
4	272.6 (0.78)	264.6 (0.78)	11.64	14.67 (4.83)	0.32 (0.80)	3.55 (0.78)	1.59 (0.78)	15.6 (0.72)	5.97 (0.75)
5	214.3 (0.61)	203.3 (0.60)	1872.49	1875.52 (618.05)	0.29 (0.73)	2.87 (0.63)	1.45 (0.71)	13.3 (0.61)	4.66 (0.58)
Armadillo (307k triangles)									
1	143.5 (1.00)	140.73 (1.00)	0.00	0.47 (1.00)	0.40 (1.00)	2.63 (1.00)	1.88 (1.00)	95.8 (1.00)	12.23 (1.00)
2	89.7 (0.62)	85.87 (0.61)	6.21	6.69 (14.06)	0.25 (0.62)	1.64 (0.62)	1.03 (0.54)	60.1 (0.62)	7.24 (0.59)
3	91.4 (0.63)	87.82 (0.62)	7.06	7.54 (15.85)	0.25 (0.62)	1.67 (0.63)	1.10 (0.58)	61.5 (0.64)	7.53 (0.61)
4	116.5 (0.81)	113.67 (0.80)	1.68	2.15 (4.53)	0.32 (0.79)	2.18 (0.82)	1.48 (0.78)	77.5 (0.80)	9.53 (0.77)
5	95.2 (0.66)	91.39 (0.64)	321.23	321.71 (675.96)	0.28 (0.70)	1.80 (0.68)	1.13 (0.60)	65.3 (0.68)	7.78 (0.63)
Hairball (2,850k triangles)									
1	2448 (1.00)	2114.84 (1.00)	0.00	5.24 (1.00)	2.04 (1.00)	17.00 (1.00)	15.23 (1.00)	476 (1.00)	110.03 (1.00)
2	1347 (0.55)	987.64 (0.46)	231.92	237.16 (45.22)	0.97 (0.47)	6.63 (0.39)	5.18 (0.34)	166 (0.34)	41.30 (0.37)
3	1362 (0.55)	1003.56 (0.47)	285.32	290.56 (55.40)	0.98 (0.48)	6.90 (0.41)	5.28 (0.35)	174 (0.36)	43.33 (0.39)
4	1892 (0.77)	1576.44 (0.74)	26.78	32.02 (6.10)	1.57 (0.76)	12.58 (0.74)	9.35 (0.62)	318 (0.66)	72.19 (0.65)
5	1473 (0.60)	1119.17 (0.52)	3053.79	3059.04 (583.31)	1.20 (0.58)	8.34 (0.50)	6.28 (0.42)	202 (0.42)	48.47 (0.43)

Table 3: *The results for individual objects where the base hierarchy for all methods is created with a spatial median top down driven build (method 1), which for methods 2 to 5 is followed by an update phase. These are as follows: 2. our method with random selection, 3. our method with the combined measure of inefficiency, 4. Kensler's method, hill climbing, and 5. Kensler's method, simulated annealing. The values in brackets are the computed ratios against reference values (1.00).*

M	cost [-]	ocost [-]	CPU update [s]	CPU build [s]	CPU primary [s]	CPU random [s]	GPU primary [ms]	GPU random [ms]	GPU occlusion [ms]
Conference (283k triangles)									
1	130.3 (1.00)	110.5 (1.00)	0.00	3.45 (1.00)	0.56 (1.00)	2.05 (1.00)	1.31 (1.00)	17.5 (1.00)	20.45 (1.00)
2	103.8 (0.79)	85.0 (0.76)	5.56	9.01 (2.60)	0.40 (0.72)	1.48 (0.72)	1.31 (1.00)	13.5 (0.77)	18.91 (0.92)
3	102.5 (0.78)	83.7 (0.75)	1.07	4.53 (1.31)	0.39 (0.69)	1.47 (0.71)	1.29 (0.98)	13.4 (0.77)	18.56 (0.90)
4	122.9 (0.94)	105.5 (0.95)	1.16	4.62 (1.33)	0.51 (0.91)	1.92 (0.93)	1.42 (1.08)	16.2 (0.92)	20.36 (0.99)
5	104.1 (0.79)	85.7 (0.77)	299.00	303.00 (87.50)	0.40 (0.71)	1.50 (0.72)	1.34 (1.02)	14.0 (0.80)	18.99 (0.92)
Fairy Forest (174k triangles)									
1	95.1 (1.00)	79.4 (1.00)	0.00	1.96 (1.00)	0.62 (1.00)	1.41 (1.00)	1.68 (1.00)	19.7 (1.00)	31.64 (1.00)
2	92.5 (0.97)	76.8 (0.96)	1.39	3.36 (1.70)	0.62 (1.00)	1.39 (0.98)	1.62 (0.96)	19.0 (0.96)	30.91 (0.97)
3	91.5 (0.96)	75.8 (0.95)	0.40	2.37 (1.20)	0.61 (0.98)	1.33 (0.94)	1.63 (0.97)	19.1 (0.97)	30.63 (0.96)
4	93.9 (0.98)	78.2 (0.98)	0.60	2.57 (1.30)	0.65 (1.04)	1.38 (0.97)	1.71 (1.01)	19.5 (0.99)	32.07 (1.01)
5	93.8 (0.98)	78.0 (0.98)	179.00	181.00 (91.80)	0.67 (1.07)	1.42 (1.00)	1.94 (1.15)	20.6 (1.04)	32.92 (1.04)
Sibenik Cathedral (80k triangles)									
1	82.3 (1.00)	71.5 (1.00)	0.00	0.66 (1.00)	0.55 (1.00)	1.19 (1.00)	1.73 (1.00)	8.7 (1.00)	25.59 (1.00)
2	70.2 (0.85)	61.6 (0.86)	1.19	1.85 (2.79)	0.48 (0.87)	1.00 (0.84)	1.42 (0.82)	7.1 (0.81)	20.93 (0.81)
3	69.5 (0.84)	61.1 (0.85)	0.72	1.38 (2.08)	0.47 (0.85)	0.96 (0.81)	1.41 (0.81)	7.1 (0.81)	22.32 (0.87)
4	77.9 (0.94)	68.7 (0.95)	0.22	0.88 (1.33)	0.55 (1.00)	1.15 (0.96)	1.72 (0.99)	8.5 (0.97)	25.48 (0.99)
5	70.8 (0.85)	62.0 (0.86)	79.12	79.79 (120.00)	0.52 (0.94)	1.00 (0.84)	1.73 (1.00)	8.3 (0.95)	25.49 (0.99)
Sponza (76k triangles)									
1	220.2 (1.00)	189.7 (1.00)	0.00	0.56 (1.00)	0.61 (1.00)	2.83 (1.00)	1.34 (1.00)	25.4 (1.00)	26.26 (1.00)
2	183.3 (0.83)	156.6 (0.82)	0.95	1.52 (2.70)	0.42 (0.70)	2.16 (0.76)	1.10 (0.82)	18.9 (0.74)	22.76 (0.86)
3	182.2 (0.82)	155.2 (0.81)	0.44	1.01 (1.79)	0.39 (0.65)	2.09 (0.73)	1.10 (0.82)	18.6 (0.73)	22.86 (0.87)
4	206.1 (0.93)	178.2 (0.93)	0.22	0.78 (1.39)	0.47 (0.78)	2.61 (0.92)	1.33 (0.99)	24.1 (0.94)	26.14 (0.99)
5	188.1 (0.85)	161.3 (0.85)	80.70	81.27 (144.40)	0.46 (0.76)	2.27 (0.80)	1.36 (1.01)	22.9 (0.89)	26.60 (1.01)
Soda Hall (2,169k triangles)									
1	216.5 (1.00)	188.9 (1.00)	0.00	40.96 (1.00)	0.64 (1.00)	1.44 (1.00)	1.13 (1.00)	11.3 (1.00)	6.02 (1.00)
2	167.1 (0.77)	141.4 (0.74)	99.41	140.00 (3.42)	0.50 (0.77)	1.16 (0.80)	0.98 (0.86)	8.6 (0.76)	4.95 (0.82)
3	165.4 (0.76)	139.7 (0.73)	15.39	56.36 (1.37)	0.50 (0.77)	1.22 (0.84)	0.96 (0.84)	8.7 (0.76)	4.94 (0.82)
4	203.1 (0.93)	177.2 (0.93)	12.53	53.49 (1.30)	0.62 (0.97)	1.41 (0.97)	1.21 (1.07)	10.8 (0.95)	6.00 (0.99)
5	180.5 (0.83)	154.8 (0.81)	2570.00	2611.00 (63.70)	0.59 (0.92)	1.38 (0.95)	1.12 (0.99)	10.6 (0.93)	5.73 (0.95)
Power Plant, section 9 (122k triangles)									
1	57.9 (1.00)	38.7 (1.00)	0.00	1.36 (1.00)	0.16 (1.00)	0.75 (1.00)	0.86 (1.00)	4.5 (1.00)	1.33 (1.00)
2	51.8 (0.89)	32.8 (0.84)	2.78	4.14 (3.04)	0.14 (0.86)	0.64 (0.85)	1.08 (1.25)	4.2 (0.92)	1.26 (0.94)
3	51.8 (0.89)	32.6 (0.84)	0.26	1.62 (1.19)	0.15 (0.88)	0.64 (0.84)	1.12 (1.30)	4.2 (0.93)	1.26 (0.94)
4	54.9 (0.94)	36.1 (0.93)	0.59	1.96 (1.43)	0.16 (0.99)	0.72 (0.96)	1.11 (1.29)	4.3 (0.96)	1.29 (0.96)
5	51.9 (0.89)	32.8 (0.84)	124.00	125.00 (91.50)	0.15 (0.89)	0.64 (0.85)	0.98 (1.13)	4.2 (0.93)	1.28 (0.96)
Power Plant, section 16 (366k triangles)									
1	93.5 (1.00)	74.1 (1.00)	0.00	4.70 (1.00)	1.20 (1.00)	0.86 (1.00)	2.98 (1.00)	5.7 (1.00)	7.60 (1.00)
2	80.2 (0.85)	61.5 (0.82)	8.71	13.41 (2.85)	1.00 (0.83)	0.72 (0.84)	2.68 (0.89)	4.8 (0.85)	6.62 (0.87)
3	79.8 (0.85)	60.9 (0.82)	2.80	7.50 (1.59)	1.07 (0.89)	0.71 (0.82)	2.93 (0.98)	4.8 (0.84)	6.83 (0.89)
4	89.3 (0.95)	70.7 (0.95)	1.29	6.00 (1.27)	1.19 (0.99)	0.81 (0.94)	2.99 (1.00)	5.5 (0.96)	7.71 (1.01)
5	83.4 (0.89)	64.9 (0.87)	420.00	424.00 (90.20)	1.14 (0.95)	0.77 (0.90)	3.25 (1.09)	5.3 (0.92)	7.40 (0.97)
Power Plant (12,748k triangles)									
1	115.8 (1.00)	85.7 (1.00)	0.00	396 (1.00)	1.55 (1.00)	0.86 (1.00)	4.52 (1.00)	6.2 (1.00)	9.00 (1.00)
2	98.3 (0.84)	68.8 (0.80)	808.00	1204 (3.04)	1.03 (0.66)	0.68 (0.79)	4.37 (0.96)	4.9 (0.79)	6.86 (0.76)
3	97.8 (0.84)	68.3 (0.79)	85.52	481 (1.21)	1.12 (0.72)	0.68 (0.79)	4.19 (0.92)	5.0 (0.80)	6.74 (0.74)
4	110.9 (0.95)	81.5 (0.95)	72.29	468 (1.18)	1.49 (0.96)	0.82 (0.95)	4.54 (1.00)	5.8 (0.94)	8.75 (0.97)
5	102.5 (0.88)	73.1 (0.85)	13468.00	13863 (35.00)	1.38 (0.88)	0.75 (0.87)	4.48 (0.99)	5.4 (0.87)	8.20 (0.91)
Pompeii Ten (5,646k triangles)									
1	252.9 (1.00)	190.5 (1.00)	0.00	102 (1.00)	0.74 (1.00)	3.57 (1.00)	3.90 (1.00)	44.0 (1.00)	19.54 (1.00)
2	218.8 (0.86)	159.1 (0.83)	267.00	369 (3.61)	0.61 (0.81)	2.81 (0.78)	3.76 (0.96)	34.2 (0.77)	18.10 (0.92)
3	218.0 (0.86)	158.2 (0.83)	77.38	179 (1.75)	0.61 (0.81)	2.92 (0.82)	3.74 (0.95)	33.9 (0.76)	18.05 (0.92)
4	237.5 (0.93)	178.6 (0.93)	25.30	127 (1.24)	0.71 (0.95)	3.35 (0.94)	3.98 (1.02)	41.3 (0.93)	19.64 (1.00)
5	225.9 (0.89)	166.6 (0.87)	6049.00	6151 (60.30)	0.67 (0.90)	3.09 (0.86)	5.25 (1.34)	39.1 (0.88)	19.19 (0.98)

Table 4: *The results of the BVH optimization for architectural scenes. The base hierarchy for all methods (M) is created with a SAH top down driven build (method 1). For methods 2 to 5 the initially constructed BVH is followed by an optimization phase using different methods and settings: 2 - our method with random selection, 3 - our method with the combined measure of inefficiency, 4 - Kensler's method, hill climbing, and 5 - Kensler's method, simulated annealing. The values in brackets are the computed ratios against reference values (1.00).*

M	cost [-]	ocost [-]	CPU update [s]	CPU build [s]	CPU primary [s]	CPU random [s]	GPU primary [ms]	GPU random [ms]	GPU occlusion [ms]
Conference (283k triangles)									
1	842.2 (1.00)	789.5 (1.00)	0.00	0.44 (1.00)	5.56 (1.00)	16.55 (1.00)	11.00 (1.00)	198.6 (1.00)	172.03 (1.00)
2	103.4 (0.12)	84.7 (0.10)	15.05	15.50 (34.76)	0.39 (0.07)	1.49 (0.09)	1.32 (0.12)	13.8 (0.06)	19.74 (0.11)
3	102.7 (0.12)	84.1 (0.10)	7.37	7.81 (17.53)	0.38 (0.06)	1.53 (0.09)	1.38 (0.12)	13.7 (0.06)	19.27 (0.11)
4	213.7 (0.25)	197.4 (0.25)	2.74	3.19 (7.16)	1.39 (0.25)	4.43 (0.27)	2.97 (0.27)	41.7 (0.21)	47.43 (0.27)
5	105.2 (0.12)	86.6 (0.10)	307.87	308.32 (691.42)	0.39 (0.07)	1.51 (0.09)	1.27 (0.11)	13.8 (0.06)	19.41 (0.11)
Fairy Forest (174k triangles)									
1	185.0 (1.00)	171.3 (1.00)	0.00	0.24 (1.00)	1.95 (1.00)	3.16 (1.00)	4.73 (1.00)	50.1 (1.00)	93.52 (1.00)
2	93.8 (0.50)	78.1 (0.45)	4.13	4.38 (17.55)	0.66 (0.34)	1.42 (0.44)	1.76 (0.37)	20.0 (0.39)	32.83 (0.35)
3	92.9 (0.50)	77.3 (0.45)	0.99	1.24 (5.00)	0.65 (0.33)	1.39 (0.44)	2.23 (0.47)	20.2 (0.40)	38.28 (0.40)
4	123.5 (0.66)	110.5 (0.64)	0.91	1.16 (4.65)	1.04 (0.53)	2.10 (0.66)	2.91 (0.61)	30.7 (0.61)	53.06 (0.56)
5	95.6 (0.51)	80.0 (0.46)	182.62	182.87 (731.61)	0.68 (0.35)	1.48 (0.46)	1.88 (0.39)	20.7 (0.41)	33.71 (0.36)
Sibenik Cathedral (80k triangles)									
1	390.7 (1.00)	380.4 (1.00)	0.00	0.09 (1.00)	4.14 (1.00)	7.61 (1.00)	10.11 (1.00)	75.0 (1.00)	188.32 (1.00)
2	71.1 (0.18)	63.0 (0.16)	2.38	2.47 (26.92)	0.50 (0.12)	1.00 (0.13)	1.50 (0.14)	7.5 (0.09)	25.17 (0.13)
3	71.3 (0.18)	63.3 (0.16)	1.07	1.16 (12.75)	0.53 (0.12)	0.98 (0.12)	1.68 (0.16)	7.3 (0.09)	27.98 (0.14)
4	121.1 (0.31)	113.7 (0.29)	0.96	1.05 (11.51)	1.43 (0.35)	2.17 (0.28)	4.04 (0.40)	17.7 (0.23)	69.68 (0.37)
5	72.6 (0.18)	63.5 (0.16)	81.62	81.71 (888.42)	0.56 (0.13)	1.01 (0.13)	1.53 (0.15)	7.6 (0.10)	26.48 (0.14)
Sponza (76k triangles)									
1	1258 (1.00)	1205.4 (1.00)	0.00	0.07 (1.00)	7.25 (1.00)	22.45 (1.00)	14.40 (1.00)	288.2 (1.00)	295.31 (1.00)
2	185.5 (0.14)	158.8 (0.13)	3.00	3.08 (39.14)	0.45 (0.06)	2.24 (0.10)	1.20 (0.08)	20.5 (0.07)	25.08 (0.08)
3	183.4 (0.14)	156.7 (0.12)	2.03	2.11 (26.81)	0.39 (0.05)	2.17 (0.10)	1.19 (0.08)	19.7 (0.06)	24.16 (0.08)
4	498.4 (0.39)	476.5 (0.39)	0.74	0.82 (10.47)	3.25 (0.44)	8.88 (0.41)	6.29 (0.44)	112.0 (0.38)	121.37 (0.40)
5	196.5 (0.15)	169.1 (0.14)	83.73	83.81 (1061.12)	0.49 (0.06)	2.49 (0.11)	1.35 (0.09)	22.5 (0.07)	28.71 (0.09)
Soda Hall (2,169k triangles)									
1	1396 (1.00)	1355.7 (1.00)	0.00	4.29 (1.00)	11.82 (1.00)	18.09 (1.00)	21.58 (1.00)	261.8 (1.00)	86.88 (1.00)
2	168.2 (0.12)	142.6 (0.10)	198.90	203.20 (47.33)	0.62 (0.05)	1.29 (0.07)	1.16 (0.05)	9.4 (0.03)	5.45 (0.06)
3	165.8 (0.11)	140.2 (0.10)	72.41	76.70 (17.91)	0.52 (0.04)	1.21 (0.06)	1.17 (0.05)	8.7 (0.03)	4.77 (0.05)
4	486.7 (0.34)	468.9 (0.34)	27.63	31.93 (7.48)	3.25 (0.28)	5.27 (0.29)	10.31 (0.48)	60.1 (0.22)	24.68 (0.28)
5	195.4 (0.13)	170.2 (0.12)	2399.95	2404.29 (560.03)	0.69 (0.05)	1.64 (0.09)	1.41 (0.06)	12.9 (0.04)	6.75 (0.07)
Power Plant, section 9 (122k triangles)									
1	188.7 (1.00)	169.6 (1.00)	0.00	0.16 (1.00)	1.03 (1.00)	3.54 (1.00)	4.95 (1.00)	22.0 (1.00)	3.28 (1.00)
2	52.1 (0.27)	32.9 (0.19)	4.04	4.21 (24.82)	0.15 (0.14)	0.65 (0.18)	1.13 (0.22)	4.3 (0.19)	1.27 (0.38)
3	52.3 (0.27)	33.1 (0.19)	2.62	2.79 (16.57)	0.15 (0.14)	0.66 (0.18)	1.21 (0.24)	4.3 (0.19)	1.28 (0.39)
4	77.8 (0.41)	60.8 (0.35)	1.05	1.22 (7.18)	0.33 (0.32)	1.43 (0.40)	1.82 (0.36)	8.3 (0.37)	1.77 (0.53)
5	51.9 (0.27)	32.8 (0.19)	128.18	128.35 (755.12)	0.15 (0.14)	0.65 (0.18)	1.21 (0.24)	4.2 (0.18)	1.27 (0.38)
Power Plant, section 16 (366k triangles)									
1	505.1 (1.00)	476.3 (1.00)	0.00	0.60 (1.00)	10.11 (1.00)	6.32 (1.00)	27.74 (1.00)	49.1 (1.00)	44.03 (1.00)
2	80.2 (0.15)	61.5 (0.12)	20.81	21.41 (35.21)	1.03 (0.10)	0.73 (0.11)	2.84 (0.10)	5.0 (0.10)	7.01 (0.15)
3	79.7 (0.15)	61.1 (0.12)	17.18	17.79 (29.28)	0.98 (0.09)	0.69 (0.11)	2.82 (0.10)	4.8 (0.09)	6.88 (0.15)
4	176.9 (0.35)	161.4 (0.33)	3.67	4.28 (7.03)	3.04 (0.30)	1.97 (0.31)	6.94 (0.25)	13.9 (0.28)	16.70 (0.37)
5	87.1 (0.17)	68.6 (0.14)	458.93	459.53 (754.25)	1.32 (0.13)	0.84 (0.13)	3.13 (0.11)	5.5 (0.11)	8.48 (0.19)
Power Plant (12.748k triangles)									
1	661.3 (1.00)	612.8 (1.00)	0	36 (1.00)	12.73 (1.00)	6.23 (1.00)	45.93 (1.00)	56.0 (1.00)	74.42 (1.00)
2	97.8 (0.14)	68.3 (0.11)	1958	1994 (55.51)	1.05 (0.08)	0.67 (0.10)	4.45 (0.09)	5.0 (0.08)	6.80 (0.09)
3	98.6 (0.14)	69.0 (0.11)	249	285 (7.92)	1.09 (0.08)	0.69 (0.11)	3.65 (0.07)	4.9 (0.08)	6.70 (0.09)
4	205.5 (0.31)	179.9 (0.29)	258	294 (8.18)	5.00 (0.38)	1.96 (0.31)	16.10 (0.35)	16.0 (0.28)	29.13 (0.39)
5	108.7 (0.16)	79.5 (0.12)	13809	13845 (385.23)	1.46 (0.11)	0.83 (0.13)	4.82 (0.10)	6.0 (0.10)	9.34 (0.12)
Pompeii Ten (5,646k triangles)									
1	766.9 (1.00)	703.8 (1.00)	0	11 (1.00)	3.00 (1.00)	17.65 (1.00)	17.79 (1.00)	276.3 (1.00)	81.87 (1.00)
2	220.3 (0.28)	160.7 (0.22)	416	427 (38.92)	0.63 (0.20)	2.92 (0.16)	4.05 (0.22)	35.3 (0.12)	18.51 (0.22)
3	220.0 (0.28)	160.3 (0.22)	105	116 (10.63)	0.63 (0.20)	2.97 (0.16)	4.27 (0.24)	34.5 (0.12)	18.98 (0.23)
4	316.5 (0.41)	268.1 (0.38)	68	79 (7.20)	1.28 (0.42)	6.08 (0.34)	7.02 (0.39)	89.1 (0.32)	31.84 (0.38)
5	229.5 (0.29)	170.4 (0.24)	6203	6214 (565.91)	0.69 (0.22)	3.20 (0.18)	4.26 (0.23)	39.7 (0.14)	19.71 (0.24)

Table 5: *The results for architectural scenes where the base hierarchy for all methods is created with a spatial median top down driven build (method 1), which for methods 2 to 5 is followed by an update phase. These are as follows: 2 - our method with random selection, 3 - our method with the combined measure of inefficiency, 4 - Kensler's method, hill climbing, and 5 - Kensler's method, simulated annealing. The values in brackets are the computed ratios against reference values (1.00).*

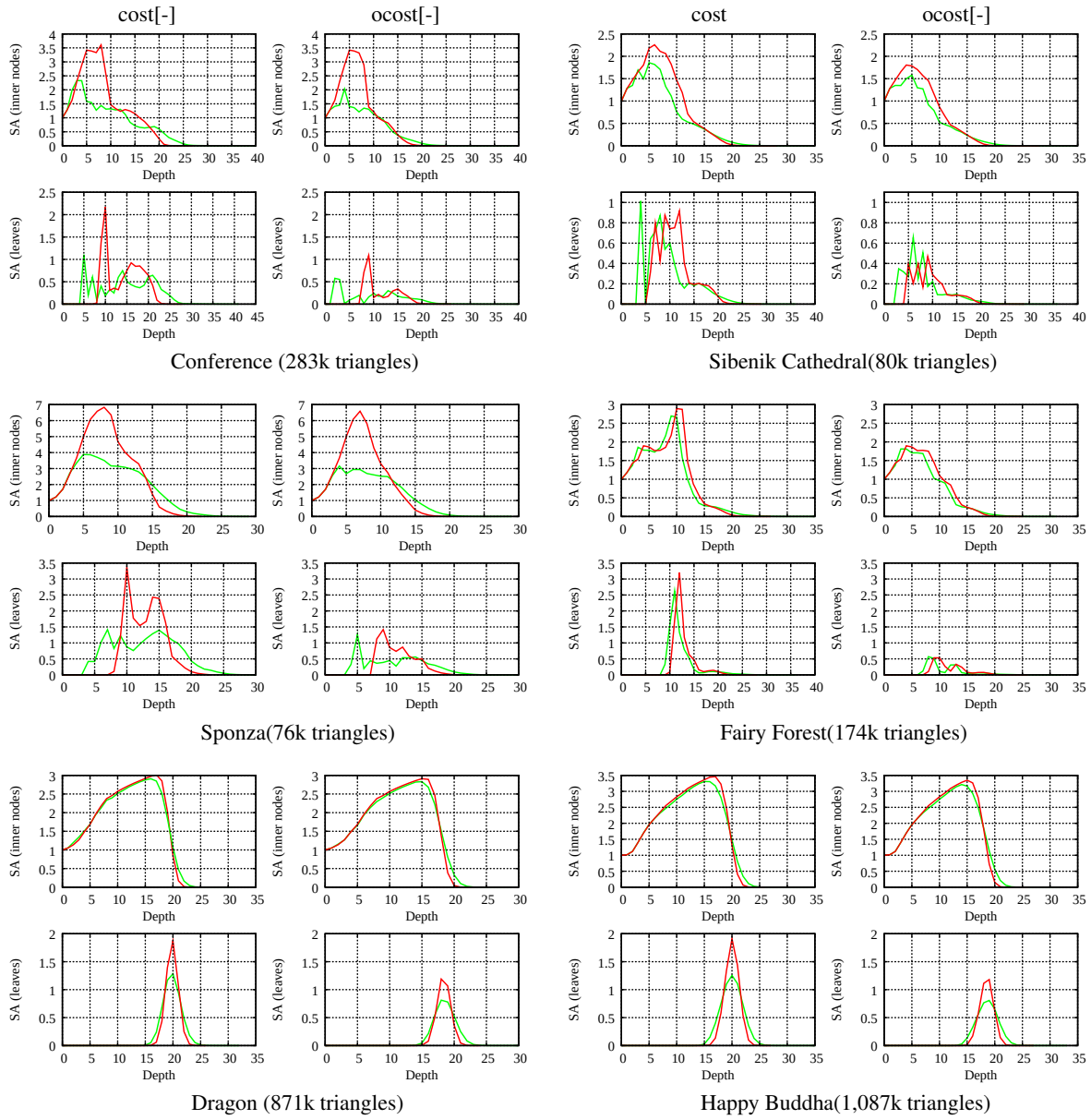


Figure 11: *Histograms showing the sums of surface areas of inner nodes and leaves at different depths of the tree. The figures in the odd columns correspond to a tree with leaves representing individual triangles (*cost *in Table 2, 3, 4, and 5), the figures in even columns to compacted trees with more triangles per leaf (*ocost *in Table 2, 3, 4, and 5). The odd rows show the sum of costs for interior nodes and the even rows the sum of costs for leaves in the dependence on the depth. The results for trees constructed by top down building algorithm with surface area heuristic are in red colour, the results for the trees after optimization by our algorithm are in green colour.*