České vysoké učení technické v Praze
Fakulta elektrotechnická

Czech Technical University in Prague
Faculty of Electrical Engineering

Ing. Jiří Bittner, Ph.D.

Optimalizované datové struktury pro sledování paprsků

Optimized Data Structures for Ray Tracing

Summary

Efficient spatial data structures are a key for achieving good performance of ray tracing based image synthesis methods. I will present an algorithm for fast optimization of bounding volume hierarchies (BVH), which became one of the most popular data structures for ray tracing. The proposed method performs 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. The method can quickly reduce the cost of the hierarchy constructed by the traditional techniques such as the surface area heuristic or spatial median splits. I will document the properties of the proposed method on scenes of different complexity. The results show that the proposed 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%.

Souhrn

Efektivní prostorové datové struktury jsou klíčovou komponentou k dosažení celkové efektivity metod syntézy obrazu založených na sledování paprsků. Představím algoritmus pro rychlou optimalizaci hierarchií obálek (BVH), které se staly jednou z nejpoužívanějších datových struktur pro sledování paprsků. Navržená metoda využívá selektivní aktualizace hieararchie založené na cenovém modelu odvozeném z povrchové heuristiky (SAH). V každém kroku algoritmus aktualizuje zlomek uzlů s cílem minimalizovat celkovou cenu hierarchie. Aktualizace jsou realizovány pomocí jednoduchých operací na uzlech stromu: odebrání, vyhledání a vložení. Metoda dokáže rychle zredukovat cenu hierarchie konstruované tradičními technikami jako je povrchová heuristika nebo dělení pomocí prostorového mediánu. Představím vlastnosti navržené metody na scénách s různou složitostí. Výsledky ukazují, že navržená metoda dokáže vylepšit hierarchii obálek konstruovanou pomocí povrchové heurstiky až o 27% a hierarchii konstruovanou pomocí prostorového mediánu až o 88%.

Klíčová slova: Viditelnost, sledování paprsků, hierarchie obálek, detekce kolizí.

Keywords: Visibility, ray tracing, bounding volume hierarchies, collision detection.

Contents

1 Introduction

The current ray tracing based algorithms allow to capture 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 work we propose a method which allows 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.

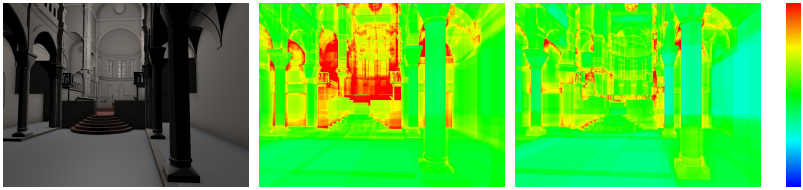

Figure 1: Visualization of the number of traversal steps for the view of the Sibenik Cathedral scene rendered with primary and ambient occlusion rays. (left) Rendered image. (center) Number of traversal steps per ray for BVH built with SAH. (right) Number of traversal steps for the BVH optimized by the proposed method. On this image our method gives an approximately 16% performance gain over the base build algorithm with the SAH. Note that the blue color corresponds to 0 traversal steps, the red color to 100 or more steps.

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 the tested 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.

There is a large body of literature on efficient spatial data structures for rendering and the bounding volume hierarchies have a long tradition in the context of ray tracing. We reffer the interested reader to the discussion of related work in the extended version of this article [BHH13].

2 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.

2.1 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 [Wal07, HSZ+11]. Alternatively we can use a faster BVH construction method based on object, or spatial median splits. 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)

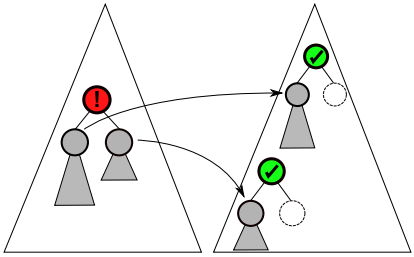

Figure 2: . The core of our method is the removal of inefficient nodes from the tree and re-insertion of their children to positions that decrease the overall cost of the tree.

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.

2.2 Cost Model

The surface area heuristic (SAH) [MB90,Hav00] is usually described using a formula evaluating the expected number of operations for processing a given ray. 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:

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

where c_T is the cost of traversing the inner node of the tree including the box intersection calculations, c_I is the cost of 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.

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 [BHH13]. Therefore reducing the cost is directly reflected in reducing ray tracing times and as we show in Section 3.

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:

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

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 of 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.

2.3 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.

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 3. Note that nodes L and R need not be leaves, but they can represent whole subtrees of the BVH.

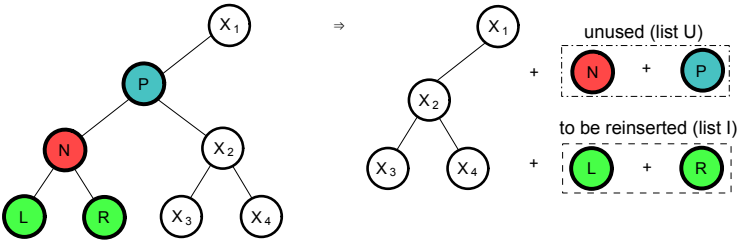

Figure 3: Illustration of removal of node N – RemoveNode(node N, list U, list I, root P) operation. The children of N are put into the list I that contains the nodes to be inserted. Nodes N and P are put to the list U that stores the nodes that can be reused.

Searching for new positions. We start the search for the new position to insert the node L at the root of the BVH and 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 4. 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.

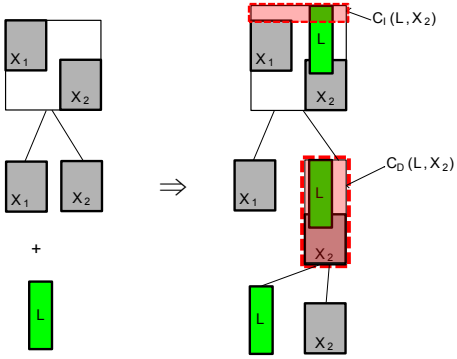

Figure 4: A 2D example of how the total cost increase of adding the node L to the tree at node X_2 is computed. The induced cost $C_I(L, X_2)$ results from enlarging ascendants of X_2 when inserting the node L as the bounding boxes have to be refitted (Algorithm 1:line 20). The direct cost $C_D(L, X_2)$ is surface area of the box for the union of X_2 and L (Algorithm 1:line 11).

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 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.

Similar cost model was used in the early work of Goldsmith and Salmon [GS87] for insertion based BVH construction. 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.

Algorithm: FindNodeForReinsertion(*node* L)
// C_{best} - the smallest total cost increase found so far C_{best} = infinity;
// Priority queue contains pairs: (node, induced cost) Push (*Root of* BVH, $0, \frac{1}{\varepsilon}$) to priority queue PQ;
while PQ *is not empty* **do**
    (X, $C_I(L, X)$) = Pop node from PQ;
    if $C_I(L, X)$ + $SA(L)$ ¿= C_{best} **then**
        // Early termination - not possible
        break; *// to reduce the cost* C_{best}
    end
    // Compute the total cost of merging L *with* X
    $C_D(L, X) = \mathrm{SA}(X \cup L)$; *// Direct cost*
    $C(L, X) = C_I(L, X) + C_D(L, X)$; *// Total cost*
    if $C(L, X)$ ¡ C_{best} **then**
        // Merging L *and* X *decreases the best cost*
        $C_{best} = C(L, X)$;
        // X_{best} *is the currently best node found*
        $X_{best} = X$;
    end
    // Calculate the induced cost for children of X
    $C_I = C(L, X)$ - $SA(X)$;
    // Check if the cost decrease is possible in subtree
    if C_I + $SA(L)$ ¡ C_{best} **then**
        if X *is not a leaf* **then**
            // Search in both children
            Push (left child of X, C_I, $\frac{1}{C_I+\varepsilon}$) to PQ ;
            Push (right child of X, C_I, $\frac{1}{C_I+\varepsilon}$) to PQ ;
        end
    end
end
return X_{best};

Algorithm 1: FindNodeForReinsertion(*node* L) - pseudo-code 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 5.

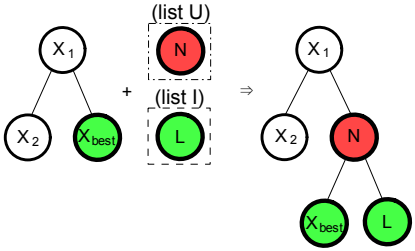

Figure 5: Illustration of reinserting the node L back to the tree. The node L is merged with the node X_{best} while using the node N as their new parent. The same procedure is used for inserting the node R (using P as the parent node).

2.4 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 a number of inefficiency measures, that address different spatial configurations of objects [BHH13]. While some of the measures are specifically designed to capture dynamic changes, we observed that one of the most stable one was the M_{AREA} measure given as $M_{AREA}(N) = SA(N)$. 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.

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.

2.5 Terminating the 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 slightly 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 deterministic inefficiency measures 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.

2.6 BVH 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 [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 3).

3 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) E5440 2.83GHz CPU, 40GB of RAM, and GNU C compiler version 4.40. To compute the cost according to equations 1 and 1 we used $c_T = 3.0$ and $c_I = 2.0$. We used batch size $k = 1\%$ and termination criteria $p_T = 10$, and $p_R = 5$. We have evaluated the method on nine models of different complexity (see Figure 6). 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.

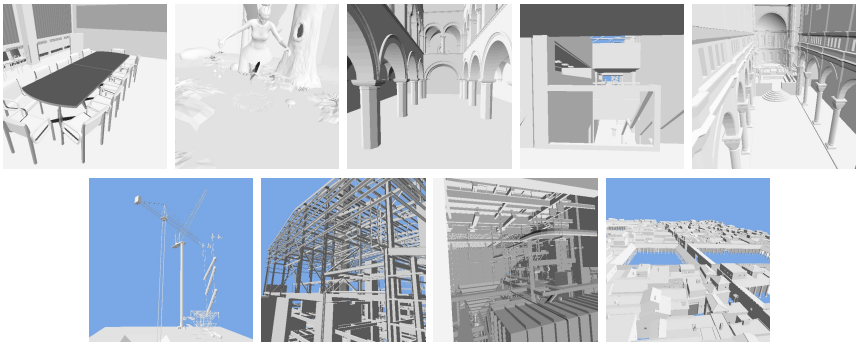

Figure 6: Snapshots of tested scenes: Conference, Fairy Forest, Sponza, Sodahall, Sibenik Cathedral, Power Plant section 9, Power Plant section 16, Power Plant, and Pompeii Ten.

3.1 BVH Cost Reduction

As a reference for the 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 2.6 (*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).

BVH cost reduction. The results show that 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 summary in Table 1). 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

Scene	cost SAH [-]	rel. cost optimized [-]	time SAH [s]	rel. time optimized [-]
Conference	130.30	78.66%	3.45	131.30%
Fairy Forest	95.10	96.21%	1.96	120.92%
Sibenik Cathedral	82.30	84.45%	0.66	209.09%
Sponza	220.20	82.74%	0.56	180.36%
Soda Hall	216.50	76.40%	40.96	137.60%
Power Plant, sec. 9	57.90	89.46%	1.36	119.12%
Power Plant, sec. 16	93.50	85.35%	4.70	159.57%
Power Plant	115.80	84.46%	396.00	121.46%
Pompeii Ten	252.90	86.20%	102.00	175.49%

Table 1: Summary results of our algorithm. The SAH-based build is used as a reference.

is about two orders 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 using a simple spatial median split the cost reduction achieved by our algorithm is very significant. 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. The time needed for the build including optimization is 25 to 147 times smaller than the time of simulated annealing by Kensler.

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 7. 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.

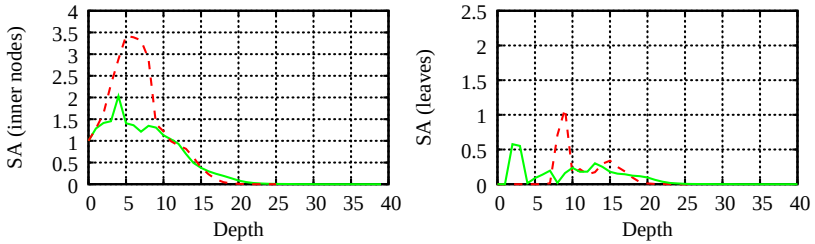

Figure 7: Histogram showing the sums of surface areas of inner nodes (left) and leaves (right) at different depths of the tree for the Conference scene. The results for trees constructed by top down SAH are shown in red (dashed), the results after optimization by our algorithm are in green (solid).

BVH tree compaction. We also evaluated the contribution of the tree compaction described in Section 2.6. The *ocost* after the compaction can be non-negligibly lower than the initial *cost* before performing the tree compaction.

The *ocost* is on average by 20% lower than *cost* for the trees originally built with SAH and by 17% for trees built with spatial median. 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).

3.2 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 6. 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. 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. 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%).

4 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.

We currently work on extending the method to dynamic and animated scenes. We also want to study the properties of the hierarchy for other visibility computations such as occlusion and view-frustum culling for large scenes.

Acknowledgements

I would like to thank Vlastimil Havran and Michal Hapala, the co-authors of the paper which presents most of the material discussed here in extended form [BHH13].

References

[BHH13] Jiří Bittner, Michal Hapala, and Vlastimil Havran. Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum*, 32(1):85–100, 2013.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

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

[HSZ+11] Qiming Hou, Xin Sun, Kun Zhou, C. Lauterbach, and D. Manocha. Memory-Scalable GPU Spatial Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):466–474, Apr 2011.

[KAL09] Tero Karras, Timo Aila, and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs; Google Code, 2009.

[Ken08] Andrew Kensler. Tree Rotations for Improving Bounding Volume Hierarchies. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 73–76, Aug 2008.

[MB90] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(6):153–65, 1990.

[Wal07] Ingo Wald. On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, pages 33–40, Sep 2007.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), January 2007.

5 Ing. Jiří Bittner, Ph.D.

Personal Information

Born:	November 9, 1972 in Brandýs nad Labem
Address:	Veltruská 532/11, 19000 Praha 9, Czech Rep.
Email:	bittner@fel.cvut.cz
Homepage:	http://dcgi.felk.cvut.cz/members/bittner
Family status:	married, two children
Languages:	Czech, English, German

Education

2003–2003	Postdoc at Vienna University of Technology
1997–2003	PhD degree in Informatics and Computer Science, Czech Technical University in Prague
1991–1997	Master degree in Computer Science, Czech Technical University in Prague
1987–1991	Secondary school (computer engineering), Prague
1979–1987	Elementary school, Prague

Citations and Impact

Google Scholar (2014)	citations: 961, h-index: 18, i10-index: 23
Research Gate (2014)	score 16.15, impact points 18.34

Professional History

2011–present	Assistant professor and deputy head for research at the Department of Computer Graphics and Interaction, FEE CTU in Prague.
2006–2011	Senior researcher at the Czech Technical University in Prague
2005–2006	Researcher at the Vienna University of Technology (Gametools IST project)
2003–2004	CGX - development of computer graphics and signal processing applications
2002–2003	Researcher at the Vienna University of Technology (UrbanViz FWF project)
2000–2002	Researcher at the Czech Technical University in Prague, Center for Applied Cybernetics

Research Activities

- *Journal reviews:* ACM Transactions on Graphics, IEEE Transactions on Visualizations and Computer Graphics, Computers & Graphics, Computer Graphics Forum, Computer Animation and Virtual Worlds.
- *Conference reviews:* SIGGRAPH, SIGGRAPH Asia, Eurographics, Graphics Interface, Symposium on Interactive 3D Graphics and Games, Pacific Graphics, Visualization, WSCG, High Performance Graphics, Eurographics Symposium on Rendering, Computer Graphics International, GRAPP.
- *IPC memberships:* Eurographics (2015), International Conference on Computer Graphics Theory and Applications (2007-2014), Winter School of Computer Graphics (2009-2014), Spring Conference on Computer Graphics (2007-2014), Center European Seminar on Computer Graphics (2007-2014), Eurographics Symposium on Rendering (2008-2010), Eurographics state-of-the-art reports co-chair (2007), Computer Graphics International (2004)
- *Organisations:* Member of Eurographics and ACM SIGGRAPH.

Selected Journal Publications

1. J. Bittner, M. Hapala, V. Havran: Fast Insertion Based Optimization of Bounding Volume Hierarchies, Computer Graphics Forum, 32(1), pages 85-100, February 2013.
2. T. Barak, J. Bittner, V. Havran: Temporally Coherent Adaptive Sampling for Imperfect Shadow Maps. Computer Graphics Forum, 32(4), pages 87-96, 2013.
3. M. Vinkler, M. Hapala, J. Bittner, V. Havran: Massively Parallel Hierarchical Scene Sorting with Applications in Rendering. Computer Graphics Forum, 32(8), pages 13-25, 2013.
4. L. Cmolik, J. Bittner: Layout-aware optimization for interactive labeling of 3D models. Computers & Graphics, vol. 34, no. 4, p. 378-387, 2010.
5. J. Bittner, O. Mattausch, P. Wonka, V. Havran, M. Wimmer: Adaptive Global Visibility Sampling, ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009), Volume 28, Issue 3, pages 94:1-94:10, 2009.

6. O. Mattausch, J. Bittner, M. Wimmer: CHC++: Coherent Hierarchical Culling Revisited. Computer Graphics Forum, vol. 27, no. 2, p. 221-230, 2008.

7. K. Zhou, E. Zhang, J. Bittner, P. Wonka: Visibility-driven Mesh Analysis and Visualization through Graph Cuts. IEEE Transactions on Visualization and Computer Graphics, vol. 14, no. 6, p. 1667-1674, 2008.

8. J. Bittner, M. Wimmer, H. Piringer, W. Purgathofer: Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful Computer Graphics Forum, 23(3):615-624, 2004.

9. J. Bittner and P. Wonka: Visibility in Computer Graphics. Journal of Environment and Planning B: Planning and Design Vol 5., No. 30, pages 729-756, Pion Ltd., 2003.

10. J. Bittner, J. Prikryl, and P. Slavik: Exact Regional Visibility Using Line-Space Partitioning Computers & Graphics, Vol. 27/4, pages 569-580, 2003.

11. J. Bittner and V. Havran: Exploiting Coherence in Hierarchical Visibility Algorithms, The Journal of Visualization and Computer Animation, Volume 12, Issue 5, pages 277-286, 2001.

12. V. Havran and J. Bittner: LCTS: Ray Shooting using Longest Common Traversal Sequences, Computer Graphics Forum, 19(3):C59-C70, 2000.

Selected Conference Publications

1. J. Bittner, O. Mattausch, A. Silvennoinen , M. Wimmer: Shadow Caster Culling for Efficient Shadow Mapping. In Proceedings of the ACM Symposium on Interactive 3D Graphics and Games (I3D'11), p. 81-88, 2011.

2. J. Bittner, V. Havran: RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In Proceedings of Spring Conference on Computer Graphics, pages 51-57, 2009.

3. J. Bittner, P. Wonka, M. Wimmer: Fast Exact From-Region Visibility in Urban Scenes. In Proceedings of Eurographics Symposium on Rendering '05, pages 223-230, 2005.