Terminating Spatial Hierarchies by A Priori Bounding Memory

Carsten Wächter Alexander Keller

Technical Report
Ulm University, 2007

We consider the efficient top-down construction of spatial hierarchies for accelerating ray tracing and present a new termination criterion that allows for a priori fixing the memory footprint of the data structure. The resulting simplification of memory management makes algorithms shorter, more efficient, and more widely applicable, especially when considering hardware implementations and kd-trees. Even the on demand construction of parts of the hierarchy becomes close to trivial. In addition it proves to be the case that the hierarchy construction time easily can be predicted by the size of the provided memory block. This in turn allows for automatically balancing construction and ray tracing time in order to minimize the total time needed to render a frame from scratch.

1 Introduction

Photorealistic image synthesis consists of connecting light sources and pixels by light transport paths and summing up their contributions. Vertices along these transport paths are found by tracing straight rays from one point of interaction to the next one. But also many other direct simulation methods in scientific computing rely on tracing particles along straight lines. Usually, a considerable part of the total computation time is spent on ray tracing.

The time spent for tracing many rays can be dramatically shortened by constructing an auxiliary acceleration data structure that allows for the efficient exclusion of large portions of the scene to be intersected with the rays instead of intersecting each ray with all objects in a scene.

We will focus on implementations of ray tracing as a backtracking algorithm that searches trees or directed acyclic graphs, because these schemes adapt best to general geometry. Constructing such a hierarchical data structure is easily formulated as a recursive procedure whose construction time is amortized when tracing many rays.

The efficiency of the ray tracing algorithm heavily depends on how the search data structures are built. Aside from various existing heuristics, memory management is an issue. While hierarchically partitioning the list of objects [WK06, WMS06, WBS06] allows one to predict the memory footprint, these techniques based on bounding vol-

umes can suffer from inefficiencies caused by large objects and the convexity of the applied bounding volumes [WK06, Wal04, Sec.7.3.3]. Partitioning space [RSH00, WIK+06, Wal04, Hav01, Ben06, Kel98, RSH05] ameliorates these problems, however, objects now can be referenced multiple times as they may intersect multiple elements of a spatial hierarchy and it had been infeasible to efficiently predict this multiplicity and thus the memory footprint in a general way.

In the following we present a solution to the memory footprint prediction problem of spatial partition hierarchies, which in addition can avoid memory fragmentation completely. This improvement applies to all previous approaches. Then we derive a memory prediction heuristic, analyze the algorithm, and present numerical experiments.

As we consider only the efficient construction of the acceleration data structure, we will not treat its efficient traversal, which however is found in [Wal04] and [Ben06]. For the following we also assume familiarity with state of the art ray tracing technology (see [Gla89, Shi00] for the roots of ray tracing).

2 Construction of Spatial Hierarchies

In order to present our new algorithm, we need to sketch the general recursive top-down procedure to construct spatial hierarchies. The steps are the following:

Termination: The termination condition is a combination of controlling memory consumption and considering the amortization of appending more nodes to the tree. Usually one clause terminates the tree by depth, which only results in an exponential bound on the number of nodes and thus memory. Another common clause avoids to further subdivide the tree if not enough references are provided. This clause saves memory and balances node versus leaf processing time.

Split selection: A heuristic is used to split the current volume element into two or more elements. Both the efficiency of the construction and the efficiency of ray tracing later on are heavily influenced by how the split is selected. Heuristics range from simple criteria as splitting axis-aligned volumes in the middle along the longest side, over global criteria [WK06] to more costly methods like the surface area heuristic (SAH) [GS87, Hav01, Wal04, WBS06]. Our new technique works with any of these heuristics.

Classification: For each object referenced in the object list it is determined which of the new volumes it intersects.

Node creation: For each new child a list of references to objects is assembled according to the classification in the previous step. The case of concern now is the reference replication caused by objects that are assigned to more than one child as it is difficult to a priori predict their number.

Post-processing: After recursively processing the children, a post-processing step can perform several tree optimizations. Examples are found in [Kel98, App. A] and [Wal04].

2.1 Termination by Bounding Available Memory

In fact only two modifications of the previous procedure are sufficient to construct a tree in an a priori fixed piece of memory.

First, we extend the argument list of the construction procedure by passing along a contiguous block of memory along with its size. Instead of terminating the hierarchy by controlling the maximum depth of the tree, a leaf node is constructed, if the two reference lists resulting from the classification step plus the size of two tree nodes do not fit into the given memory block.

Second, we predict the memory consumption of the two new sub-trees in the sorting step and accordingly schedule the available memory to the two children when continuing recursively. We therefore compute a prediction $p \in [0,1]$, which schedules the fraction of memory for the left child. The remainder of the memory is scheduled for the right child. Note that for branches with more than two children we need a prediction $p_i \in [0,1]$ for each child i with the condition that all p_i sum up to one.

Contrary to the classical clause, which terminated the tree by one global depth parameter, the depth now is implicitly controlled by the scheduled memory. The scheduling allows to locally adapt the depth, which is superior to the previous exponential bound. Scheduling in addition replaces the second classical termination clause, which limited the number of items worth a further tree subdivision. The same effect now can be obtained by just providing a smaller memory block upon the initial call of the construction procedure.

The procedure succeeds as long as there is at least sufficient memory to store one node along with the list of references to all objects in the scene. This is a reasonable assumption as exactly this data must be provided upon each procedure call anyhow.

The construction algorithm now can be implemented to run on limited memory and in place (see the illustration for a binary tree in Figure 1). Instead of single memory allocations for nodes and lists, the memory management reduces to one block allocation before the initial call of the construction routine.

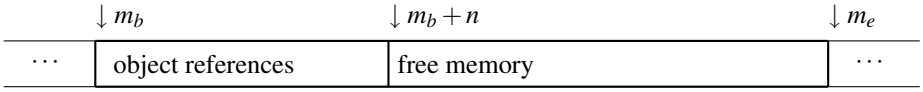
Being able to fit a hierarchy into an a priori fixed memory footprint meets the requirements of a hardware implementation, too, where a limited amount of on-board memory is a fact.

2.2 Scheduling Available Memory by Prediction

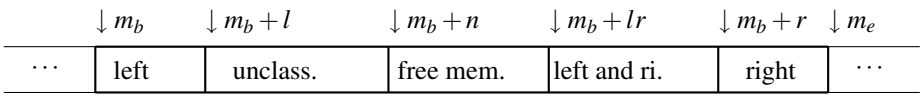
In order to predict the ratios p_i, we first take a look at bounding volume hierarchies [WBS06, WK06, WMS06], which recursively partition the list of objects and thus lack reference replication. Built to full depth, the number of inner nodes in the tree plus one is equal to the total number of objects, which is true for any subtree, too. As a consequence the relative frequency

$$p_i = \frac{e_i}{\sum_{j=1}^{m} e_j} \tag{1}$$

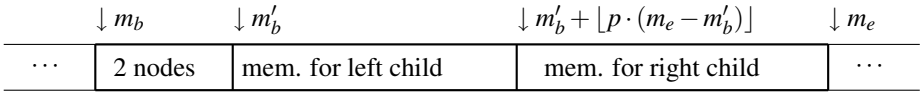
a) Upon procedure call



b) Indices during in-place sorting



c) Creation of children



d) Alternative memory layout in pre-order

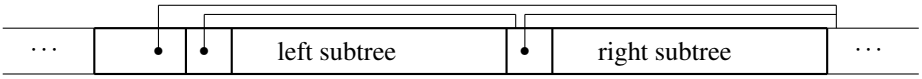


Figure 1: Illustration of a contiguous block of memory for a binary kd-tree. a) Upon a call the construction procedure is provided the memory block starting at m_b and ending at m_e along with the number n of objects. b) In order to enable in-place sorting the array is partitioned into regions for objects either on the left or right, or in both children, the unclassified items to be processed, and the remaining free memory area. See the text for the complete algorithm description. c) If children are created, the memory for two nodes is taken from the provided memory block and the remainder starting at m'_b is scheduled for the two children according to the prediction $p \in [0,1]$. d) A memory layout in pre-order improves cache performance, because less memory is accessed.

determined by the count e_i of objects to be sorted into the i-th child of an m-ary node, respectively, exactly predicts the fraction of memory to be scheduled for the i-th subtree.

If the tree is not built to full depth, the prediction remains optimal in the sense that all subtrees will receive memory proportional to their number of objects. This means that all leafs will store about equal numbers of objects as they become about uniformly pruned and subtrees containing more objects will be deeper. To summarize, our new termination criterion along with proportional prediction allows one to build efficient bounding volume hierarchies in a predetermined contiguous block of memory.

2.3 Prediction in the Presence of Reference Replication

There are situations where bounding volume hierarchies suffer from severe performance penalties: For example these are encountered when bounding boxes of objects expose big regions of overlap, when axis-aligned boxes are used to bound non-convex objects, or when empty space cannot be separated as e.g. for diagonal pipes [Wal04, Sec.7.3.3]. Then, instead of partitioning object lists, usually spatial partition schemes are applied.

Considering hierarchies such as e.g. kd-trees [SS92, Kel98, Wal04, Ben06] that potentially replicate references to objects that intersect splitting planes, we still can use the prediction in equation (1) to schedule the memory for the children. This simple prediction overcomes the static termination by tree depth as it does not just cut the deepest branches of the tree but uniformly bounds memory proportional to the number of references in a subtree as mentioned before.

2.3.1 Prediction Discrepancy

However, the prediction in equation (1) relies on local information only and cannot predict the reference replication in the lower levels of the tree. Hence there may be a discrepancy between the prediction and the optimal subtree memory footprint. As a consequence some subtrees can be pruned by insufficient memory, while others cannot use the scheduled memory. Although this effect vanishes as more memory becomes available, it strongly implies that reference replication should be minimized by e.g. preferring split planes aligned to the bounding boxes of the objects [HKRS02] in order to allow for potentially deeper trees and consequently more efficient culling.

Our experiments indicate that performance degradations are hardly noticeable, which is strongly supported by [WH06]: The $\mathcal{O}(n_t \log n_t)$ behavior allows to draw the conclusion that the observed average case reference replication is at most linear in the number n_t of objects, which is a perfect justification of the proportional heuristic.

However, from theory [MR95] a worst case behavior of $\mathcal{O}(n_t^2)$ is known and thus, although maybe rare, situations must exist, where the prediction will not work well: The number of replicated references is proportional to the surface area of the object. Long, thin objects will create references proportional to the length of their longest side, while more extended objects will create references quadratic in the longest side. As long as objects are small with respect to the total scene size, this effect will not be

noticeable as it happens only in some levels of the hierarchy. For a Mikado/Jackstraws-game like scene, however, the effect will be quite noticeable.

Another situation where proportional prediction will fail locally is a prediction of $p = 0.5$, where all objects sorted to the left are randomly placed, whereas the objects on the right describe regular structures. While the right subtree will have minimal reference replication the left one will have a lot, however, both subtrees were given the same amount of memory. Situations like this are not theoretical: Trees around a building are a setting, where locally the prediction can be non-optimal.

Many obvious ideas for improving the prediction will not work: For example scheduling memory proportional to surface area or volume will dramatically fail, because identical measures will schedule identical memory for the branches, although their number of objects may differ vastly. Note that this is independent of how the splits are selected (see Section 2). Solutions to this problem must aim at predicting the reference replication much higher in the hierarchy than they actually happen. The optimal prediction can be found by iterating the hierarchy construction: The replication counts from a previous tree can be used to predict the memory for the next tree. This approach is especially promising for animations of deformable objects, where only geometry is transformed.

3 Complexity Analysis and Memory Footprint

The construction procedure coincides with a quicksort. While it is easy to see that finding the pivot element in fact corresponds to selecting a splitting plane, it is rather unconventional that some elements are generated by reference replication during sorting. However, their maximum number is bounded by the a priori fixed memory footprint.

The analysis of the quicksort algorithm is readily available in standard textbooks on algorithms and data structures and states an average $\mathcal{O}(n \log n)$ running time in the number of n of object references to be sorted. This matches the observations in [WH06]. The actual running time depends on how the split is selected and on how the references are ordered. Of course there exist geometric configurations which violate the assumptions of [WH06] and cause the worst case running time of $\mathcal{O}(n^2)$ as can be seen in the initial segment behavior of the graphs documented in [WH06]. This is in accordance with the theoretical results in [MR95].

The question is now how to choose the size n of the memory footprint. It must be sufficiently large to store at least one node along with all object references. But there are more choices and considerations:

1. Providing memory $n = \alpha \cdot n_t$ proportional to the number of objects to be ray traced is more reasonable. The factor $\alpha > 1$ then represents the amount of allowed reference replication (see also Figure 2). This is the most practical choice and was actually motivated by the findings in [WH06], because an $\mathcal{O}(n \log n)$ quicksort can only touch $\mathcal{O}(n)$ memory.

2. Choosing $n = \beta \cdot n_r$ proportional to the number n_r of rays to be shot, which in some path tracing algorithms is proportional to the number of pixels, exposes an interesting relation to the classical Z-buffer: The memory footprint is linear in

the number of pixels, however, only if the memory required to store the scene objects does not exceed this limit. As will be illustrated in the Section 5 on numerical experiments (see also Figure 3), we thus are able to amortize the hierarchy construction time by controlling it by the number of rays shot. An extreme example is a single ray, where no hierarchy needs to be built and the triangles are just tested in linear order.

3. Providing the maximum available memory reduces the risk of penalties from bad memory predictions. However, building the tree adaptively then can result in subtrees incoherently scattered over the whole memory block. Even worse, the construction of *k*d-tree can fill arbitrary amounts of memory due to reference replication. Consequently this last option is a bad choice.

Besides these obvious sizes and combinations thereof, we are convinced that there exist other useful methods. Note that ray tracing complexity is not only determined by the cost of constructing hierarchy; in fact the complexity later on is ruled by the backtracking search, as indicated by the second item of the above enumeration. The parameters α and β can be used to adjust the amortization (see Section 5).

4 Implementation

The new concept is easily verified by just implementing the termination condition using any existing implementation at hand. Although the memory allocation remains untouched, the memory footprint implicitly becomes controlled by the termination criterion.

A more efficient implementation still follows the outline of Sections 2 and 2.1, however, memory management can be simplified: Before calling the construction routine a memory block is allocated in which a root node along with the object references is stored in sequential order (similar to Figure 1a). Inside the routine the classification uses the memory layout as illustrated in Figure 1b: For the next unclassified object we decide, whether it belongs to the left, right, or both children in the subtree. The first case just requires to increment the l variable. For the second case the last element of the left-and-right block is moved to the front to make space for the new element on the right. In addition the last unclassified element is moved to the vacancy left by the element classified right. The last case requires to replicate a reference: The current element is moved to the front of the left-and-right-block and again the last unclassified element needs to be moved to the just created vacant memory position. If not enough memory is available for replication, the routine has to create a leaf node by copying back the right block and left-and-right block.

The creation of the children will only be performed, if there was sufficient memory for two nodes and the total number of references including the replicated ones. According to the memory layout in Figure 1c, some elements from the left list have to be moved to its end to make space for the nodes and the left-and-right block has to be copied to the end of the left list. Based on the classification we can use the proportional heuristic in equation (1) to compute the offset $m'_b + \lfloor p \cdot (m_e - m'_b) \rfloor$ of memory

scheduled for the right subtree. Then the memory block of the left-and-right with the only-right items has to be moved to this offset.

Using this memory layout, the optimization of sharing the left-and-right block in leaf nodes [Kel98, App. A] comes in handy and allows one more level in the tree without the cost of additional memory.

While the above description considered *k*d-tree construction, it applies as well to bounding volume hierarchies by just omitting the part that allows for reference replication. It is obvious, too, how to generalize the algorithm for *m*-ary trees.

Taking a closer look reveals memory fragmentation, unless the memory scheduled for a subtree is completely used. This is trivially avoided by recognizing the prediction p as an upper bound on the memory scheduled to the left child and just linearly writing the tree data structure into the memory. This in addition increases data coherency which is beneficial with modern processor cache architectures.

Proceeding that way may seem unfair, as the rightmost branch potentially receives the most memory, however, this can be compensated by modifying p. A simple improvement is to change the order, in which the children are built recursively. This can be done randomly or by changing the order if the tree depth is odd.

4.1 Alternative Memory Layout

The classic technique of storing the children as an array (see Figure 1c) allows one to use only one pointer. Storing the tree in pre-order (see Figure 1d) allows one to save memory: A pointer followed by its subtree points to the next subtree [Smi98, GM03]. While leaf nodes are directly followed by references to objects or the objects themselves, an inner node is followed by e.g. the splitting plane offset. This saves one level of indirections and results in more cache coherent memory access, but did not not perform noticeably different from the classic layout. In addition it does not allow for the last level optimization possible in *k*d-trees (see Section 2).

Especially for a hardware design it would be beneficial to have two separate memory areas with each a separate memory controller for the inner and leaf nodes, respectively. The ray tracing implementation in the Intel Performance Primitives uses such a layout: First the tree nodes are enumerated, which are followed by the object references. With our algorithm it is also possible to schedule a chunk of nodes in order to increase cache performance. Ideally this would be in van der Emde Boas layout, however, a penalty is paid if nodes remain empty.

4.2 Applied Split Plane Heuristics

As now both approaches of partitioning space (e.g. *k*d-trees) and partitioning object lists (e.g. bounding volume hierarchies) fit a unified framework, it becomes straightforward to use a hybrid approach to optimize performance. The split selection then tries to first divide the list of objects unless this is inefficient and reference replication by spatial partition becomes unavoidable. The corresponding ray traversal must distinguish between nodes with and without reference replication.

In anticipation of that step and with the data layout in Figure 1b, bounding volume hierarchies in this paper are built by first sorting all objects that overlap a potential split-

ting plane into the left-and-right block. After scanning all objects, it is decided whether all objects in that block are appended to either the left or right block. Compared to single decisions for each object as described in [WK06] the overlap is minimized much more. Consequently empty volume is maximized and in turn the overall performance improves.

4.3 Massive Scenes

When a scene description does not fit into main memory, we just can rely on virtual memory mechanisms of the operating system to manage the memory block [WDS04]. In order to reduce page thrashing, the upper part of the inner nodes of the spatial hierarchy can be stored in a part of memory that permanently remains in main memory [WSBW01, DH05, WK06].

We also considered two separate memory blocks for the upper part and lower parts of the hierarchy. The new termination criterion can efficiently fit the upper part of a hierarchy into the a priori fixed first memory block. The lower parts are built on demand in the second block of memory. The least recently used parts of the hierarchy become flushed, if memory is not sufficient. This procedure then somewhat resembles multi-pass algorithms on rasterization hardware.

4.4 Construction on Demand

Instead of recursively constructing the whole hierarchy in advance, the hierarchy can be built on demand during ray traversal. Therefore nodes are checked for being finalized upon visit. If they are not finalized the construction procedure is called that either finalizes the current subtree or appends one level of nodes. Therefore non-finalized nodes have to store the temporary information necessary to proceed the construction procedure upon the next visit. If the scheduled free memory does not allow to store this information, the recursive construction routine is called to build the subtree fitting in that memory block.

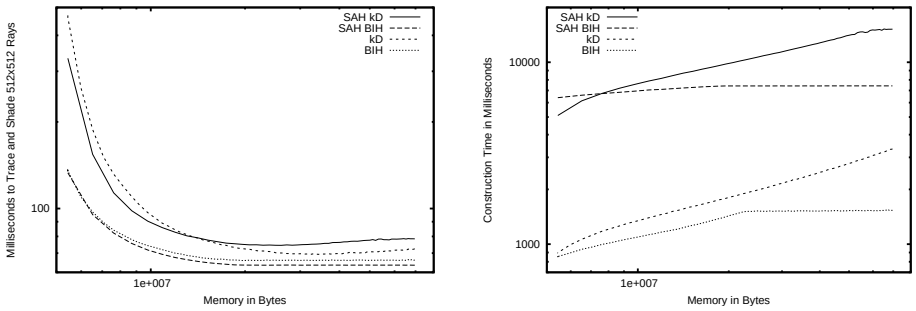
While the implementation is rather simple and elegant, it only saves computation time, but leaves unused memory regions, as the memory block is allocated once before.

5 Numerical Experiments and Results

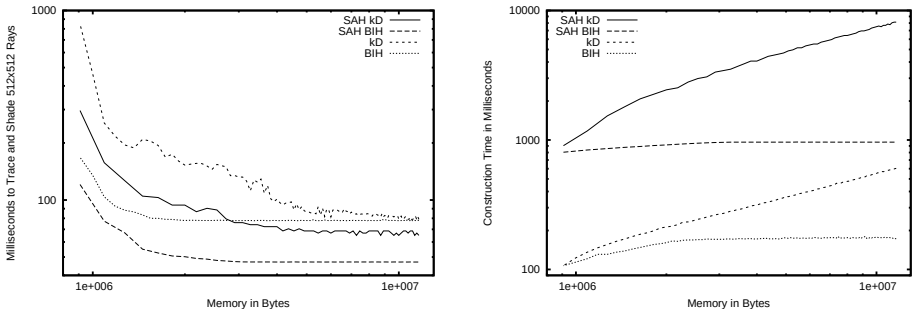
It is important to note that our termination criterion is not a new algorithm to build hierarchies, it is much more a way to efficiently control memory consumption. Also note that providing sufficient memory results in hierarchies identical to the ones built using the classic criteria.

We verified the new termination criterion using the *k*d-tree and the bounding interval hierarchy (BIH, [WK06]) as they are amongst the currently most competitive hierarchies. Measurements were done using one processor core on a Core2Duo Laptop running at 1.66GHz.

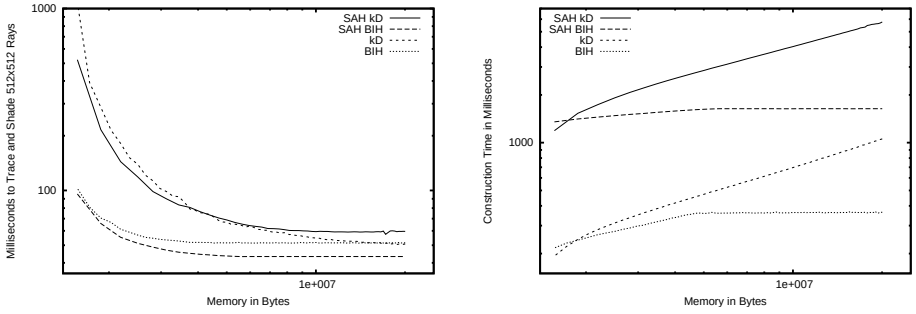
For the first set of numerical experiments three scenes have been used: A car scene represents a currently very attractive domain for ray tracing, a kitchen that has a mix-



Stanford Happy Buddha, $n_t = 1.087.716$ triangles.



TBP Kitchen, $n_t = 181.755$ triangles.



Car 1, $n_t = 312.888$ triangles.

Figure 2: The graphs illustrate the asymptotic behavior of our new termination criterion. For each of the three test scenes the memory ranges from $\alpha = 5 \ldots 64$ bytes times the number n_t of triangles. The left graph shows how the ray tracing time (excluding construction time) for rendering 512×512 pixel images depends on the size of the memory available for the acceleration data structure. As expected an exponential decay is observed. The right graph shows how limiting the memory first behaves linear in the number of triangles and as the trees become deeper asymptotically blends to an $\mathcal{O}(n \log n)$ complexity. Since the bounding interval hierarchy (BIH) construction does not replicate references, the BIH curves become clipped once the tree is built to the maximally possible depth. Note that using the SAH efficiently requires additional memory linear in the number n_t of triangles, whereas the graphs only consider the memory used for the actual acceleration data structure.

ture of small and big partially overlapping triangles, and the Stanford Happy Buddha as a very simple scene. For our measurements we applied four split plane heuristics: The *k*d-trees were built by splitting through the vertex nearest to the middle of the longest axis of the axis-aligned voxel and by an SAH approximation using 20 candidates on each axis [SSK07]. The bounding interval hierarchies (BIH) were using the global heuristic as described in [WK06] including the refinement from Section 4.2 as well as the full SAH [WH06]. The BIH SAH implementation variant requires additional precomputation time and memory for the presorted lists (16 bytes times the number of triangles in the scene) to achieve $\mathcal{O}(n \log n)$ running time. Yet, the additional memory consumption is omitted in the graphs, otherwise all BIH SAH plots must be thought of shifted to the right. In Figure 2 we illustrate the behavior of our new termination criterion on the size of the provided memory block. Providing more memory the hierarchies can be built deeper and as expected the time spent for ray tracing decreases. Looking precisely, it can be seen that for the Buddha ray traced with a *k*d-tree too much memory allows for too deep trees that in turn become inefficient. The construction time asymptotically behaves like $\mathcal{O}(n \log n)$ on the average. The beginning segments of the construction time curve, however, are linear in the number of triangles as not enough memory is provided to build deep hierarchies. Contrary to the *k*d-trees the BIH curves are clipped, because no reference replication is possible.

In the second set of experiments we used the Stanford Thai statue to investigate how the construction time can be amortized over rendering time, i.e. how the total time needed to render a frame from scratch can be minimized. From Figure 3 it becomes obvious that this minimum depends on both the number of rays and the memory block provided. In other words: The quality of the hierarchy needs to be traded for its construction time. While the "valley" of the minimum is clearly visible for the *k*d-tree in Figure 3, it is less distinct for the BIH (although similar), because the BIH can be constructed much faster due to the lack of checks for reference replication. The shape of the graphs in Figure 3 is easily explained: With increasing resolution first the construction time dominates before the ray tracing time takes over. The more rays are shot the deeper, i.e. better, hierarchies pay off. This depth is controlled by the new termination criterion and thus by the size of the available memory.

The above observation together with our new termination criterion is especially useful in a dynamic setting, where each frame must be computed from scratch. Here the frame rate is easily maximized by measuring the total time to image and then increasing or decreasing the size of the provided memory block from frame to frame. This allows for automatically determining a close to optimal memory block size for offline animation rendering, too.

6 Conclusion

We introduced a new termination criterion and memory scheduling heuristic that allows one to construct an efficient ray tracing acceleration data structure in an a priori fixed memory block. The resulting simplified memory management is beneficial for both software and especially hardware implementations.

Although we gave strong arguments for the proportional memory scheduling heuris-

tic, still other heuristics could be developed and explored. The principle to terminate hierarchies by memory consumption applies to other ray tracing acceleration schemes, too, as for example hierarchical grids [WIK+06, JW89, CDP95, KS97], octrees, or even ray classification [AK87]. It is straightforward to apply the scheme to classic BSP trees (see e.g. [FvDFH96] or current games) in order to determine visibility and to point clouds (e.g. photon mapping [Jen01]) for faster range searching. Finally it is interesting to explore the new scheme in the field of collision detection and occlusion culling with graphics hardware.

Acknowledgements

The authors would like to thank mental images GmbH for support and for funding of this research. Special thanks go to Manuel Finckh and Johannes Hanika for their quick help with the SAH measurements.

References

[AK87] J. Arvo and D. Kirk, *Fast Ray Tracing by Ray Classification*, Computer Graphics (Proc. SIGGRAPH 1987) **21** (1987), no. 4, 55–64.

[Ben06] C. Benthin, *Realtime Ray Tracing on current CPU Architectures*, Ph.D. thesis, Saarland University, 2006.

[CDP95] F. Cazals, G. Drettakis, and C. Puech, *Filtering, Clustering and Hierarchy Construction: a New Solution for Ray-Tracing Complex Scenes*, Computer Graphics Forum (Proc. Eurographics 1995) **14** (1995), no. 3, 371–382.

[DH05] T. Driemeyer and R. Herken (eds.), *Programming mental ray, 3rd ed.*, Springer, 2005.

[FvDFH96] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics, Principles and Practice, 2nd Edition in C*, Addison-Wesley, 1996.

[Gla89] A. Glassner, *An Introduction to Ray Tracing*, Academic Press, 1989.

[GM03] M. Geimer and S. Müller, *A Cross-Platform Framework for Interactive Ray Tracing*, Tagungsband Graphiktag der Gesellschaft für Informatik (2003), 25–34.

[GS87] J. Goldsmith and J. Salmon, *Automatic Creation of Object Hierarchies for Ray Tracing*, IEEE Computer Graphics & Applications **7** (1987), no. 5, 14–20.

[Hav01] V. Havran, *Heuristic Ray Shooting Algorithms*, Ph.D. thesis, Czech Technical University, Praha, Czech Republic, 2001.

[HKRS02] J. Hurley, R. Kapustin, A. Reshetov, and A. Soupikov, *Fast Ray Tracing for Modern General Purpose CPU*, Proc. Graphicon, 2002, pp. 255–261.

[Jen01] H. Jensen, *Realistic Image Synthesis Using Photon Mapping*, AK Peters, 2001.

[JW89] D. Jevans and B. Wyvill, *Adaptive Voxel Subdivision for Ray Tracing*, Proc. Graphics Interface, 1989, pp. 164–172.

[Kel98] A. Keller, *Quasi-Monte Carlo Methods for Photorealistic Image Synthesis*, Ph.D. thesis, University of Kaiserslautern, Germany, 1998.

[KS97] K. Klimaszewski and T. Sederberg, *Faster Ray Tracing Using Adaptive Grids*, IEEE Computer Graphics & Applications **17** (1997), no. 1, 42–51.

[MR95] M. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[RSH00] E. Reinhard, B. Smits, and C. Hansen, *Dynamic Acceleration Structures for Interactive Ray Tracing*, Proc. 11th Eurographics Workshop on Rendering, 2000, pp. 299–306.

[RSH05] A. Reshetov, A. Soupikov, and J. Hurley, *Multi-Level Ray Tracing Algorithm*, ACM Transactions on Graphics (Proc. SIGGRAPH 2005) **24** (2005), no. 3, 1176–1185.

[Shi00] P. Shirley, *Realistic Ray Tracing*, AK Peters, Ltd., 2000.

[Smi98] B. Smits, *Efficiency Issues for Ray Tracing*, Journal of Graphics Tools **3** (1998), no. 2, 1–14.

[SS92] K. Sung and P. Shirley, *Ray Tracing with the BSP-tree*, Graphics Gems III (D. Kirk, ed.), Academic Press, 1992, pp. 271–274.

[SSK07] M. Shevtsov, A. Soupikov, and A. Kapustin, *Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes*, Computer Graphics Forum (Proc. Eurographics 2007) **26** (2007), no. 3, to appear.

[Wal04] I. Wald, *Realtime Ray Tracing and Interactive Global Illumination*, Ph.D. thesis, Saarland University, 2004.

[WBS06] I. Wald, S. Boulos, and P. Shirley, *Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies*, ACM Transactions on Graphics **26** (2006), no. 1.

[WDS04] I. Wald, A. Dietrich, and P. Slusallek, *An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models*, Rendering Techniques 2004 (Proc. 15th Eurographics Symposium on Rendering), 2004, pp. 81–92.

[WH06] I. Wald and V. Havran, *On building fast kD-trees for Ray Tracing, and on doing that in O(N log N)*, Proc. 2006 IEEE Symposium on Interactive Ray Tracing, September 2006, pp. 61–69.

[WIK+06] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. Parker, *Ray Tracing Animated Scenes using Coherent Grid Traversal*, ACM Transactions on Graphics (Proc. SIGGRAPH 2006) (2006), 485–493.

[WK06] C. Wächter and A. Keller, *Instant Ray Tracing: The Bounding Interval Hierarchy*, Rendering Techniques 2006 (Proc. 17th Eurographics Symposium on Rendering) (T. Akenine-Möller and W. Heidrich, eds.), 2006, pp. 139–149.

[WMS06] S. Woop, G. Marmitt, and P. Slusallek, *B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes*, Proc. Graphics Hardware, 2006, pp. 67–77.

[WSBW01] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, *Interactive Distributed Ray Tracing of Highly Complex Models*, Rendering Techniques 2001 (Proc. 12th Eurographics Workshop on Rendering), 2001, pp. 277–288.

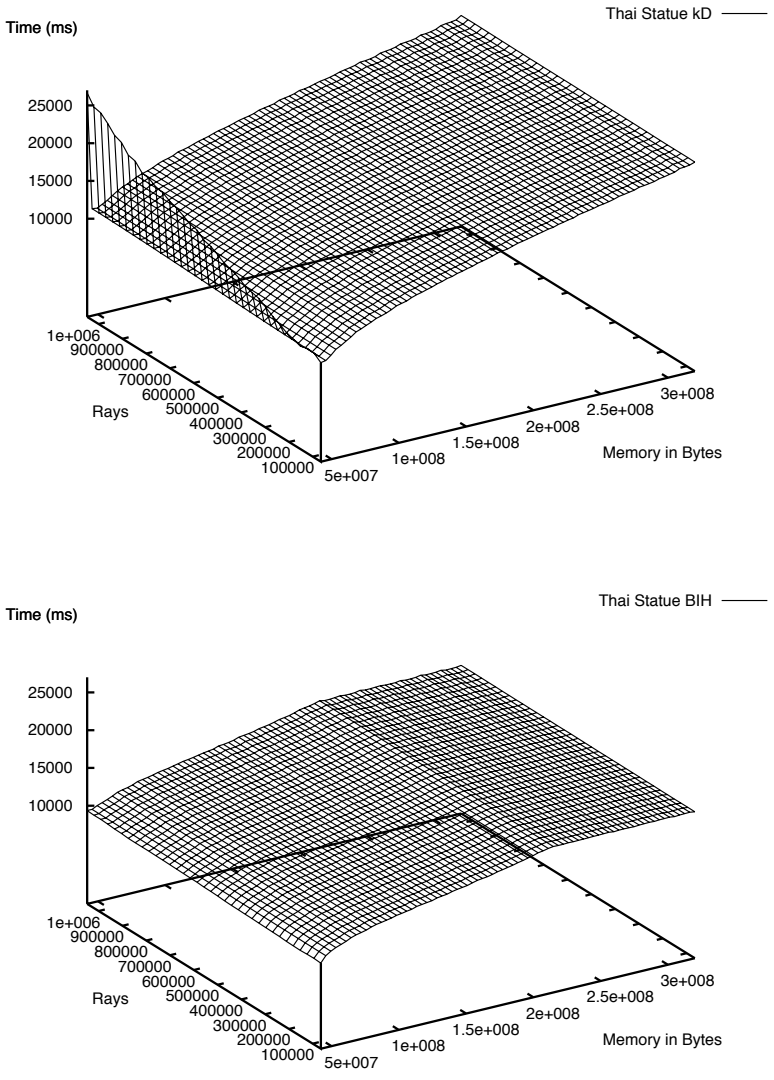


Figure 3: Illustration of the total time to image in dependence on the provided memory size and number of rays ($256^2 \dots 1024^2$) for the Stanford Thai Statue ($n_t = 10.000.000$ triangles). For both approaches the minimal time clearly depends on both the number of rays and provided memory.