Experiences with Streaming Construction of SAH KD-Trees

Stefan Popov* MPI Informatik
Johannes Günther* MPI Informatik
Hans-Peter Seidel* MPI Informatik
Philipp Slusallek† Saarland University

ABSTRACT

A major reason for the recent advancements in ray tracing performance is the use of optimized acceleration structures, namely kd-trees based on the surface area heuristic (SAH). Though algorithms exist to build these search trees in $O(n \log n)$, the construction times for larger scenes are still high and do not allow for rebuilding the kd-tree every frame to support dynamic changes.

In this paper we propose modifications to previous kd-tree construction algorithms that significantly increase the coherence of memory accesses during construction of the kd-tree. Additionally we provide theoretical and practical results regarding *conservatively* sub-sampling of the SAH cost function.

Keywords: Kd-tree construction, streaming memory access, parallelization, bounded variation function.

Index Terms: I.3.6 [Computer Graphics]: Methodology and Techniques Realism—Graphics data structures and data types I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1 INTRODUCTION

Ray tracing has evolved in the last years into a real-time technique for image synthesis. The current state of the hardware allows real-time ray tracing to be carried out even on a single computer [15, 22]. A significant factor – besides the faster hardware – that contributed to the lately witnessed speedup in ray tracing performance was the introduction of more effective acceleration structures and enhanced traversal algorithms. In particular, kd-trees have been observed to give superior performance in ray tracing compared to other acceleration structures [5], especially when built using the surface area heuristic (SAH) [12].

Unfortunately, construction of optimized kd-trees is expensive and currently does not allow for interactive ray-tracing of large dynamic scenes. As ray-tracing is used more and more as a prime method for image synthesis, it becomes increasingly desirable to be able to ray trace dynamic scenes interactively as well.

There are basically three strategies for interactive ray tracing of dynamic scenes. First, one can avoid the costly re-build of kd-trees in the spirit of [11, 23] or [3, 4]. Second, one can optimize other acceleration structures for ray tracing that can be quickly built or updated, e.g. grids [14, 26], bounding volume hierarchies (BVHs) [10, 24], or hybrid data structures [21, 27]. And third, one can speed-up and optimize the *construction* of (SAH based) kd-trees. In this paper we follow the last strategy.

Our goal is to construct the kd-tree from scratch every frame without any pre-knowledge of the scene. Consequently the proposed algorithm is very flexible and does not imply any restrictions on the dynamic scene: Primitives can be added or removed between the frames, the connectivity as well as the topology can be changed, and triangles can undergo completely random motion.

*e-mail:{popov,guenther,hpseidel}@mpi-inf.mpg.de
†e-mail:slusallek@graphics.cs.uni-sb.de

In this paper we analyze the construction of SAH based kd-trees and propose several techniques to accelerate it. Using a streaming algorithm to build kd-trees we maximize memory coherence. Additionally we show that the cost function can be coarsely sampled *without* loosing accuracy. Finally we propose a parallel kd-tree construction algorithm.

2 PREVIOUS WORK

So far, research on building kd-trees has mainly focused on improving the efficiency of the resulting acceleration structure for ray tracing, i.e., how well the kd-tree can cull triangles from being tested for intersection with a ray.

For a detailed description on building kd-trees and an analysis of state-of-the-art construction algorithm see [25] by Wald and Havran. Although they propose an $O(n \log n)$ algorithm – the theoretical lower bound for comparison-based sorting algorithms – they did not apply low-level optimization techniques or other approaches to speed-up the method.

Very recently, Havran et al. [6] analyzed the problem of fast construction of different spatial hierarchies (including the kd-tree). They also proposed a SAH-based hybrid acceleration structure that can be constructed quickly by exploiting the discretization of the whole 3D domain.

Concurrent work by Hunt et al. [7] also aims at fast construction of kd-trees. By adaptive sub-sampling they approximate the SAH cost function by a piecewise *quadratic* function. Because they can afford only a relative small number of samples (16) per split the resulting kd-tree can be of sub-optimal performance for ray tracing. Although we approximate the cost function only by a piecewise *linear* function the degradation in quality of our constructed kd-trees can be neglected, because we use many more samples (1024).

Some other publications scrutinize construction of kd-trees as well, but only marginally. Benthin [1] implemented a fast kd-tree builder using SIMD instructions and he successfully demonstrated parallelization with two treads. A memory efficient variant to split the triangle list during kd-tree construction was proposed by Wächter and Keller [21]. They also report build times orders of magnitude faster than [25], but this fast algorithm is not published (yet). Finally, Stoll et al. [18] highlighted the benefits of closely coupling the scene graph and the acceleration structure for the quick (and lazy) construction of kd-trees. However, their implementation did not show this potential yet.

3 CONSTRUCTION OF SAH BASED KD-TREES REVISITED

Before we describe our contributions we need to shortly explain the state-of-the-art kd-tree construction algorithm and its cost function, because this forms the basis to understand our contributions.

3.1 The Surface Area Heuristic

The surface area heuristic [2, 12, 13, 19] provides a method to estimate the cost of a kd-tree for ray tracing. Minimizing this expected cost during construction of a kd-tree results in an approximately optimal kd-tree which provides superior ray tracing performance.

The ray tracing costs are modelled by SAH in the following way. Assuming uniformly distributed rays, the probability P_{hit} that rays will hit some sub-volume $V \subset V_S$, given that the rays hit the volume

V_S, is related to the surface area SA of these volumes [16]:

$$P_{hit}(V|V_S) = \frac{SA(V)}{SA(V_S)} \tag{1}$$

This is only correct if we further assume that the rays are actually infinite lines without start and end point.

The expected cost C_R for a random ray R to intersect a kd-tree node N is given by the cost of one traversal step K_T, plus the sum of expected intersection costs of its two children, weighted by the probability of hitting them. The intersection cost of a child is locally approximated to be the number of triangles contained in it times the cost K_I to intersect one triangle. We name the children nodes N_l and N_r and the number of contained triangles n_l and n_r, respectively. Thus we get

$$\begin{aligned} C_R &= K_T + K_I \left[n_l P_{hit}(N_l|N) + n_r P_{hit}(N_r|N) \right] \\ &\overset{(1)}{=} K_T + \frac{K_I}{SA(N)} \left[n_l SA(N_l) + n_r SA(N_r) \right] \end{aligned} \tag{2}$$

Finally, the expected cost C_T of a complete tree T can be computed as

$$C_T = K_T \sum_{N \in Nodes} \frac{SA(V_N)}{SA(V_S)} + K_I \sum_{L \in Leaves} \frac{SA(V_L)}{SA(V_S)} n_L, \tag{3}$$

where V_S is the axis-aligned bounding box (AABB) of the complete scene, and n_L the number of triangles in leave L.

3.2 Conventional $O(n \log n)$ Construction

Typically kd-trees are built in a top-down manner by recursively splitting the tree into two sub-trees, or correspondingly, splitting a kd-tree cell into two sub-cells. The expected intersection cost is calculated for all possible split locations and the one with minimal cost is used for splitting. If the ratio between this minimal cost and the cost of intersecting a ray with all triangles in the node is above some threshold, the node is made a leaf and the recursion stops.

The cost function (2) is evaluated by sweeping the bounding box of the current node N with three planes perpendicular to the x-, y- and z-axis. Because the cost function is piecewise linear it needs only to be evaluated at the boundaries of the AABBs of (the parts of) the triangles that lie inside N. These locations are also called split candidates or events.

The cost function is evaluated incrementally during the sweep by keeping track of the number of triangles intersecting the left and right sub-volumes for each split event. The sweep requires the events of each dimension to be sorted.

Sorting the events for each split results in a $O(n \log^2 n)$ construction algorithm. To achieve a better time complexity of $O(n \log n)$, the event lists are sorted only *once* at the beginning of the construction. Then the events are sifted to the two new nodes keeping their order. The distribution process works as follows (see [25]):

1. The triangles of the current node are classified as belonging to the left, right or both children using the event list corresponding to the current split dimension.

2. All triangles that will belong to both children are intersected with the splitting plane. Two new AABBs are created for each triangle – for the parts in the left and right child nodes. Finally, new temporary event lists are created from these AABBs and they are sorted afterwards.

3. The actual sifting takes place: Events for triangles belonging to only one child are moved directly, and the the temporary events from the previous step are merged – this accounts for the ignored events of straddling triangles.

For reasonable scenes a splitting plane will hit at most $\sqrt{n'}$ triangles, with n' being the number triangles in the current node [25]. Thus the build time stays $O(n \log n)$, despite the sorting in step 2.

An alternative of the algorithm works with the AABBs of the triangles only. Thus, the expansive intersection with the triangles and the splitting plane as well as the sorting of the lists in step 2 can be avoided. This variation of the algorithm is known as the boxed kd-tree builder. Although a tree built in such manner degrades the ray tracing performance somewhat (about 10 % on average, up to 35 % [5, Section 4.10.4]) a boxed builder is much faster than a non-boxed one.

4 Fast Construction of kd-Trees

4.1 Streaming Construction

Todays hardware architectures expose a clear gap between the bandwidth achieved through sequential and random memory accesses – a typical computer system can access it's main memory an order of a magnitude faster using a sequential pattern. The gap is even larger (more than two orders of magnitude) for external memory storage devices such as a hard disk. Thus, memory access patterns become an important characteristic of todays algorithms.

Locality is another important factor. Algorithms that tend to access data that has recently been accessed benefit greatly from caches and memory hierarchies. Thus random access patterns hardly hurt the performance of algorithms that preserve locality.

The conventional KD tree construction algorithm can easily become bandwidth limited for large input models, due to it's random memory access pattern in the triangle classification stage.

To overcome this limitation we developed a streaming construction algorithm that closely reassembles a boxed kd-tree builder. It constructs the tree in a BFS manner up to a certain tree level and than switches to conventional DFS building. This algorithm processes whole tree levels at one step at the beginning. The AABBs of the triangles are stored in a continuous memory array. Each node from the current level is associated with a partition from that array. Some AABBs might be duplicated in the array, because they belong to more than one node. The algorithm sweeps the array once to estimate the best split position for each partition/node by sampling. Then the algorithm creates the nodes for the next level, as all split positions for the nodes in the current one are already known. The array of AABBs is swept once again and the objects are copied into a second array. Depending on the spatial relation of the AABB to the split plane of the node, it is either copied to the partition of the left node, to the one of the right node, or clipped and copied to both children's partitions. If the algorithm encounters a partition with size bellow some threshold, the subtree corresponding to it is built using the conventional algorithm. The partition is then discarded from further processing. In the next level, the second array becomes the input array of AABBs.

The new algorithm has several advantages over the conventional one:

- It accesses memory in a sequential pattern only, except when the conventional build routine is invoked. However, the latter only operates on a small subset of the data and thus preserves locality.

- It postpones the events sorting to a later stage and performs it on small sets of data. Thus, more efficient algorithms such as radix sort can be used because all accessed data will fit into the caches of the CPU.

The cost function needs to be sampled along all three dimensions. The value of the cost function at a sampling position depends solely on the count of AABBs that intersect the left and right subvolumes for the latter location. The algorithm collects this data in two phases.

In the first phase, the algorithm sweeps the array of AABBs and stores at each sample the count of AABBs that end between this and the previous sample. The algorithm also stores the number of AABBs that start between the sample and the next one. Although the sample locations could be chosen arbitrarily it is better to distribute them uniformly, because then the sample index an event belongs to can be calculated directly from the event's position, avoiding a more expensive binary search.

In the second phase, the algorithm uses the collected information to incrementally reconstruct the AABBs counts at the sample locations using the formula

$$n_l^{i+1} = n_l^i + S^i,$$
$$n_r^{i+1} = n_r^i - E^i,$$

where $n_{l|r}^i$ describes the number of AABBs to the left/right of the sample i, and the number of AABBs that start and end between samples i and $i+1$ is denoted by S^i and E^i, respectively.

The algorithm evaluates the cost function at the sample locations and than distributes the AABBs to the left and right subtrees as described above. To save an extra sweep over the array of AABBs we already sample the cost functions of the left and right subtree during the copy. A compact representation of our streaming construction algorithm is given in Algorithm 1.

In our implementation we use 1024 regularly distributed samples for all three dimensions. The start and end indexes of the sampling locations that encompass the bounding boxes as well as the counter reconstruction in the second phase are calculated using vector SSE instructions. Note that the runtime of the function sampling is almost independent of the number m of sampling positions, because only the very cheap second phase depends on m and usually $m \ll n$.

4.2 Conservative Cost Function Sampling

By testing our implementation with many different scenes, we observed that using 1024 regularly distributed samples results in a very good approximation of the best split position. However, for some scenes it might be necessary to be able to find an arbitrary close to the exact one approximated minimum. We start by analyzing the cost function.

For one dimension, let v denote the position of a candidate split location for the current node N, which is bounded by v_{min} and v_{max}. Further let s_1 and s_2 be the extent of node N in the other two dimensions. For the best split we are looking for v that minimizes $C(v)$ with

$$C(v) = K_T + C_l(v) + C_r(v),$$
$$C_l(v) = K_I \cdot n_l(v) \frac{2(s_1+s_2)(v-v_{min}) + 2s_1s_2}{SA(N)},$$
$$C_r(v) = K_I \cdot n_r(v) \frac{2(s_1+s_2)(v_{max}-v) + 2s_1s_2}{SA(N)},$$

where $n_l(v)$ and $n_r(v)$ denote the number of objects to the left and right side of v, respectively.

The cost function can be seen as a difference of two monotonically increasing functions, hence it is a bounded variation function (see Figure 1). We can exploit this property to increase the quality of the approximation of the cost function by adaptively resampling only the intervals that can contain the minimum. For each interval $[a,b]$, we can derive the lower bound $B^l_{a,b} = K_T + C_l(a) + C_r(b)$ and the upper bound $B^u_{a,b} = K_T + C_r(a) + C_l(b)$ of the cost function. Because the values of the cost function are known at the sample points we then can also set the overall upper bound B^u to be the minimum of all upper bounds $B^u_{a,b}$. All intervals with a lower bound $B^l_{a,b} \geq B^u$ cannot contain the minimum and can thus be skipped during resampling. For most scenes, there are only few (about 3) intervals that need to be resampled (for the upper levels of the kd-tree). To get the *exact* minimum this resampling process can be repeated with very few iterations until there is only one split location left in the interval which contains the minimum.

The same technique can also be applied in event space for the conventional build algorithm. Instead of fixing the sampling positions, the cost function can be evaluated at every Nth event. Afterwards, the intervals with a lower bound for the cost function greater

Algorithm 1: Streaming Construction

```
 1: procedure UpdateSampleStatistics(aabb, statistics)
 2:     l_xyz ← indexes of samples just below min point of aabb
 3:     u_xyz ← indexes of samples just above max point of aabb
 4:     for all dim ∈ {x,y,z} do
 5:         Increase statistics.objStart[l_[dim]]
 6:         Increase statistics.objEnd[u_[dim]]
 7:     end for
 8: end procedure
 9:
10: function GetSplitLocation(stat)
11:     stat.oLeft[0] ← 0
12:     stat.oRight[0] ← #objects
13:     for all i ∈ 1..len(stat) do
14:         stat.oLeft[i] ← stat.oLeft[i−1] + stat.objStart[i]
15:         stat.oRight[i] ← stat.oRight[i−1] − stat.objEnd[i]
16:     end for
17:     Evaluate the cost function at stat
18:     return The best found split location at stat
19: end function
20:
21: aabbIn ← bounding boxes of all triangles
22: stat ← 0
23: UpdateSampleStatistics(∀aabb ∈ aabbIn, stat)
24: levelNodesIn ← {root_node, GetSplitLocation(stat)}
25:
26: while levelNodesIn ≠ ∅ do
27:     nextLevelAABB ← ∅
28:     nextLevelNodes ← ∅
29:     for all {node, splitData} ∈ levelNodesIn do
30:         if #objects in node < threshold then
31:             Run conventional build routine for subtree
32:         end if
33:         curAABBIn ← node's partition of aabbIn
34:         len_r, len_l ← #objects in the subtrees of node
                                              ▷ taken from splitData
35:         Allocate len_l + len_r space at end of nextLevelAABB
                                  ▷ first len_l elements assigned to left child
36:         stat_l ← 0, stat_r ← 0
37:         for all aabb ∈ curAABBIn do
38:             Add aabb → left child's partition in nextLevelAABB
                    ▷ if not completely to the right of the split plane
                                              ▷ clip aabb if necessary
39:             Do the same for the right child's partition
                        ▷ if not completely to the left of the split plane
40:             Update stat_L and/or stat_R
                                  ▷ invoking UpdateSampleStatistics
41:         end for
42:         Create nodes N_l, N_r for the two subtrees
43:         nextLevelNodes ←+ {N_l, GetSplitLocation(stat_l)}
44:         nextLevelNodes ←+ {N_r, GetSplitLocation(stat_r)}
45:     end for
46:     levelNodesIn ← nextLevelNodes
47:     aabbIn ← nextLevelAABB
48: end while
```

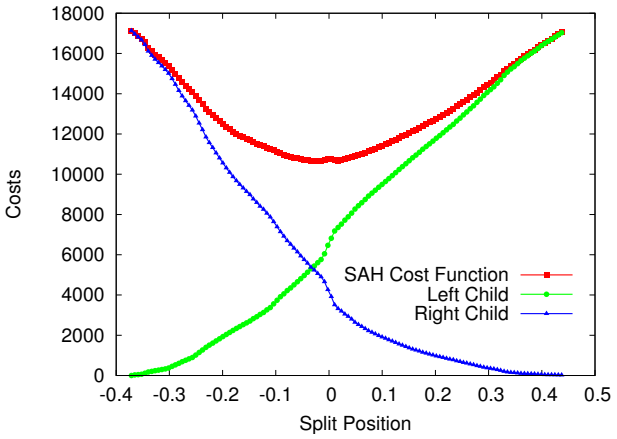

Figure 1: Example of the cost function (red) for the HAND for one of the first splits. According to SAH this cost function is the sum of two monotonic functions, corresponding to the costs of the left (green) and the right child (blue), respectively. We are interested in finding the minimum of this function to place the optimal splitting plane.

than the estimated minimum can be rejected and the cost function can be evaluated conservatively in the remaining intervals.

4.3 Improving the Conventional Construction Algorithm

Some improvements can also be achieved in the conventional build part of the algorithm.

Due to the small working set on which the conventional build algorithms operates, all memory accesses will happen in the caches of the processors. Thus, it becomes feasible to use a more efficient sorting algorithm – namely radix sort [9, 17]. The time complexity of radix sort is linear, as opposed to the $O(n \log n)$ complexity of comparison-based sorting algorithms such as quick sort or heap sort. The disadvantage of radix sort, the introduction of random memory access patterns, is neutralized by the locality of the data set. We apply an improved radix sort algorithm to the floating point coordinates of the events for the initial sorting, taking the binary representation of the floating point numbers as keys for radix sort. Negative floats need to be pre-processed to preserve the correct ordering. Additionally the complexity of the radix sort is further reduced as described in [20].

In contrast to [25], we keep three separate lists of events – one for each dimension. The cost function is evaluated by sweeping the lists of events, incrementally changing the number of objects to the left and right of the event, and finally calculating the SAH cost at the event's position. For any given coordinate value, the end events must be processed before the start events to guarantee a correct cost function value (see [25]). The events are sorted based on their coordinates only. Thus, the algorithm needs to count all start and end events for a given split position. It then subtracts the end event count from the current number of objects on the right, calculates the cost function and finally adds the count of start events to the number of objects on the left.

The evaluation of the cost function can also be optimized. We tried two strategies: conservative sub-sampling in event space, and a reduced-operation incremental approach.

For sub-sampling in event space we evaluated the cost function only at every k event position. By setting $k = 16$ at the higher levels of the kd-tree and $k = 2$ at the lower levels we adapt to the decreasing number of events per node. Unfortunately, we were not able to achieve a speedup using the sub-sampling method here, because the ratio of skipped cost function evaluations gets smaller and smaller further down in the tree. Additionally, we found that even skipping only every other event position near the leaves already significantly deteriorates the quality of the resulting kd-tree.

The second approach tries to minimize the number of operations needed to calculate the cost function. Therefore we rewrite the cost function as

$$C(v) = K_T + 2K_I \frac{s_1 + s_2}{SA(N)} \left[n_l(v)(K_l + v) + n_r(v)(K_r - v) \right],$$
$$\text{with} \quad K_l = \frac{s_1 s_2}{s_1 + s_2} - v_{min},\ K_r = \frac{s_1 s_2}{s_1 + s_2} + v_{max}.$$

This formulation of the cost function enables a better incremental update of the costs. First, instead of evaluating the full cost at each event position, only $n_l(v)(K_l + v) + n_r(v)(K_r - v)$ needs to be evaluated. Later, the real value of the cost function can be reconstructed by multiplying with a proper constant and adding the traversal cost.

If an event has the same coordinates as the preceding one, it only changes the *number* of objects to the left and right of this position. Thus, the cost can be incrementally update by adding $K_l + v$ for a start event, or by subtracting $K_r - v$ for an end event. Again, the costs of start events are temporarily accumulated and not added to the SAH-costs until all end events of this position have been processed. In the case that the split location changes between events from v to v', the cost should additionally be updated by adding $(v' - v) \cdot (n_l(v) - n_r(v))$. Incremental cost function updates require less arithmetic operations than non-incremental ones.

4.4 Parallelization

A trivial but powerful extension of the presented streaming algorithm is it's ability to run in parallel. It is possible to assign different processors to the different subtrees and build them in parallel. This is particularly useful for the lower levels where the algorithm reverts to the conventional building scheme. In our experiments, we observed that 90 % of the construction time is actually spent there. Also, since the different processors will work mostly in their caches, there will be few resource conflicts in the memory controller. By carefully managing the memory allocations and the copy operations for the subtrees, the algorithm can be implemented in a way that the data for the different subtrees stays in memory local to the processor that works on it, which is important for NUMA shared memory architecture systems.

An alternative approach to parallelization, which will also work well for the streaming part of the algorithm is to break the data in blocks, have every processor work on a separate block and create the sampling statistics for it. Then the statistics can be merged in order to evaluate the split location.

For our experiments we implemented the trivial parallelization approach. We do the initial cost function evaluation for the first level of the tree on a single processor. During the objects redistribution in the BFS part, each thread needs to know exactly where the partitions for the left and right children of the current node are located in memory. Therefore, partial sums need to be built over the partition sizes, which in the current implementation is also done on a single thread.

5 Results and Discussion

We tested our implementation with a variety of scenes (see Figure 2). Except for the triangle count, these scenes differ also in regularity. The BUNNY, BLADE, and THAI STATUE models can be found in the Stanford 3D Scanning Repository and were acquired using 3D scanning. Therefore these three scenes consist of regular sized and uniformly distributed triangles, that are also pre-sorted. In contrast, the HAND, FAIRY FOREST, CONFERENCE room, and the BEETLE model were designed by hand or with CAD tools. In particular the last three scenes have an irregular triangle distribution and contain empty space which should be exploitable by the SAH.

Figure 2: The scenes used for testing and evaluation of our streaming kd-tree construction algorithm. From left to right: HAND, BUNNY, FAIRY FOREST, CONFERENCE, BLADE, BEETLE, and THAI STATUE.

To measure the impact of the pre-sorting we additionally ran our kd-tree construction algorithm on the same scenes, but with randomly shuffled triangles.

All measurements were performed on a workstation equipped with 4 GB main memory and two dual-core AMD Opteron CPUs clocked at 2.6 GHz.

5.1 Construction Timings

To evaluate the performance of our streaming construction algorithm we compared it to the conventional construction algorithm by measuring the time needed to build a SAH kd-tree for the test scenes.

Table 1 shows that several factors influence the benefit of streaming construction. Firstly, the number of triangles determines the memory requirements and thus the cache efficiency. For smaller scenes it is more likely that irregular and incoherent memory accesses will be compensated by the caches. Therefor the improvement using the streaming construction over the conventional algorithm is not as high as for larger scenes.

Secondly, pre-sorted scenes additionally improve memory locality in favor for the conventional kd-tree construction. However, with the shuffled scenes the streaming approach clearly outperforms the conventional construction with a speedup of up to 50 %. Furthermore we provide strong evidence that our streaming kd-tree construction is independent of the order of the input data, as shuffling the triangles has hardly any effect on the construction times – even for large scenes. This is important because we cannot (and should not) assume any meaningful sorting for dynamic scenes.

Finally, we see that the construction times also depend on the type of the scene and the distribution of the triangles. For example the BLADE and the BEETLE have roughly the same number of triangles. Still, to build a kd-tree for the BEETLE takes about twice the time because of the more irregular triangles.

Additionally, we profiled the different parts of our construction algorithm, which revealed that building the lower level subtrees consumes most of the processor time. Unfortunately, the streaming approach does not help anymore at this phase. Furthermore, we observe that the time spent in re-distribution the events to the left and right sub-trees already dominates the construction time. Thus sub-sampling the cost function in the conventional build, too, does not much improve on the construction time. This is because the incremental evaluation of the SAH cost function at every candidate location is already quite cheap.

5.2 Approximative Cost Function Evaluation

We also estimated the approximation error introduced by sub-sampling the SAH cost function in the streaming splits, because this influences the quality of the built kd-trees. Therefore Table 1 also provides the expected cost according the SAH (Equation (3)) for ray tracing using the built kd-trees. Additionally, we also evaluated the average intersection cost for a ray by intersecting uniformly generated rays with the scene using the generated kd-trees. As the expected costs and the measured intersection costs were strongly correlated we excluded these latter numbers.

	1 CPU	2 CPUs		4 CPUs	
model	time (ms)	time (ms)	speedup	time (ms)	speedup
HAND	103.8	110.9	0.94	104.8	0.99
BUNNY	518.4	420.2	1.23	417.4	1.24
FAIRY FOREST	1,151	944.8	1.22	947.2	1.22
CONFERENCE	1,405	926.6	1.52	741.0	1.90
BLADE	7,603	4,686	1.62	3,395	2.24
BEETLE	15,310	9,412	1.63	6,674	2.29
THAI STATUE	81,301	49,458	1.64	33,446	2.43

Table 2: Scalability in number of CPUs of our parallelized streaming construction algorithm. We achieve a decent speedup using more processors, and again with larger scenes construction scales better. However, the scalability is still sub-linear because full parallelization including the first splits is not implemented yet.

For our test scenes the sub-sampling approach during streaming construction hardly decreased the quality of the kd-trees. Because of the very low approximation of at most 2.2 % we do not use the conservative cost function estimate refinement as proposed in Section 4.2.

5.3 Scalability with Number of CPUs

Because the trend in todays computing hardware is towards multi core systems parallel algorithms become increasingly interesting. To test the scalability of our parallelized construction algorithm we ran it on a on a 4 core system. Table 2 reports the results of our measurements. With 4 CPUs we achieve a speedup of up to 2.5 times compared to using only one CPU. The speedup is again related to the number of triangles in the scenes and is higher with larger models. Scalability is still only sub-linear because several parts – such as the first splits – of the algorithm are not yet parallelized. However, we believe that a fully parallel implementation of the streaming algorithm (including careful management of memory local to the processor) is able to scale roughly linearly with the number of cores.

6 CONCLUSION AND FUTURE WORK

In this paper we proposed a construction algorithm for SAH kd-trees that significantly improved locality of memory accesses by introducing streaming computation. Additionally we showed that the approximation error introduced by sub-sampling the cost function in the first (streamed) splits is negligible. We also provided background on how to sub-sample conservatively. An important advantage of the proposed algorithm is its potential to be used in lazy kd-tree building, because it does not require the expensive $O(n \log n)$ sorting step at the beginning as the conventional builder.

Our proposed streaming algorithm should be ideally suited for hardware implementations and the Cell [8] processor. Thus we like to implement our kd-tree builder on the Cell to evaluate its potential. Additional we want to demonstrate that the full parallelization (including the first splits) exhibits linear scalability in the number of CPUs.

model	triangles	Conventional Construction time (ms)	Conventional Construction expected cost	Streaming Construction time (ms)	Streaming Construction speedup	Streaming Construction expected cost	Streaming Construction increase
HAND	17,135	108.7	69.17	103.0	5.48 %	69.17	0.00 %
HAND shuffled	”	111.7	”	105.6	5.71 %	”	”
BUNNY	69,451	610.6	94.11	513.2	19.0 %	95.05	1.00 %
BUNNY shuffled	”	621.6	”	520.8	19.4 %	”	”
FAIRY FOREST	174,117	1,318	75.84	1,151	14.5 %	76.68	1.12 %
FAIRY FOREST shuffled	”	1,448	”	1,182	22.6 %	”	”
CONFERENCE	282,664	1,546	78.08	1,412	9.50 %	79.34	1.61 %
CONFERENCE shuffled	”	1,830	”	1,479	23.7 %	”	”
BLADE	1,765,388	7,522	157.7	7,604	-1.08 %	158.9	0.75 %
BLADE shuffled	”	10,405	”	8,038	29.5 %	”	”
BEETLE	1,873,389	18,824	70.23	15,324	22.8 %	71.73	2.14 %
BEETLE shuffled	”	22,032	”	15,598	41.3 %	”	”
THAI STATUE	10,000,000	99,556	136.7	80,849	23.1 %	138.3	1.14 %
THAI STATUE shuffled	”	122,978	”	83,083	48.0 %	”	”

Table 1: Comparison between the conventional kd-tree construction algorithm and our new streaming construction algorithm in building time and quality of the resulting kd-tree. The larger the scene, the higher the speedup using streaming construction because for the conventional build the working set does not fit into the caches anymore. Additionally, the conventional build heavily depends on pre-sorted input whereas streaming construction is largely independent of the ordering of the triangles. The increased expected cost by only maximal 2 % due to the approximative evaluation of the cost function introduced by streaming construction can be well tolerated.

Furthermore the domination costs of building the last levels of the kd-tree as well as the high costs of actually moving the events in memory during each split deserve more attention.

ACKNOWLEDGEMENTS

The authors wish to thank Carsten Benthin for his help with low-level code optimization.

REFERENCES

[1] C. Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, 2006.

[2] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[3] J. Günther, H. Friedrich, H.-P. Seidel, and P. Slusallek. Interactive ray tracing of skinned animations. *The Visual Computer*, Aug. 2006. doi: 10.1007/s00371-006-0063-x (Proceedings of Pacific Graphics).

[4] J. Günther, H. Friedrich, I. Wald, H.-P. Seidel, and P. Slusallek. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3), Sept. 2006. (Proceedings of Eurographics).

[5] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[6] V. Havran, R. Herzog, and H.-P. Seidel. On fast construction of spatial hierarchies for ray tracing. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, Sept. 2006.

[7] W. Hunt, G. Stoll, and W. Mark. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, Sept. 2006.

[8] IBM. The Cell project at IBM research. http://www.research.ibm.com/cell/, 2005.

[9] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, second edition, 1998.

[10] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. RT-DEFORM interactive ray tracing of dynamic scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, Sept. 2006.

[11] J. Lext and T. Akenine-Möller. Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001 – Short Presentations*, pages 311–318, 2001.

[12] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. In *Graphics Interface Proceedings 1989*, pages 152–163, Wellesley, MA, USA, June 1989. A.K. Peters, Ltd.

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

[14] E. Reinhard, B. Smits, and C. Hansen. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering*, pages 299–306, Brno, Czech Republic, June 2000.

[15] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Transaction of Graphics*, 24(3):1176–1185, 2005. (Proceedings of ACM SIGGRAPH).

[16] L. Santalo. *Integral Geometry and Geometric Probability*. Cambridge University Press, 2002.

[17] R. Sedgewick. *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching*. Addison-Wesley, 1998. (3rd Edition).

[18] G. Stoll, W. R. Mark, P. Djeu, R. Wang, and I. Elhassan. Razor: An architecture for dynamic multiresolution ray tracing. Technical Report TR-06-21, The University of Texas at Austin, Department of Computer Sciences, 2006. (available at http://www-csl.csres.utexas.edu/users/billmark/papers/razor-TR06/).

[19] K. R. Subramanian. *A Search Structure based on kd-Trees for Efficient Ray Tracing*. PhD thesis, University of Texas at Austin, Dec. 1990.

[20] P. Terdiman. Radix sort revisited, 2000. http://codercorner.com/RadixSortRevisited.htm.

[21] C. Wächter and A. Keller. Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006, Proceedings of the Eurographics Symposium on Rendering*, pages 139–149, Aire-la-Ville, Switzerland, June 2006. The Eurographics Association.

[22] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[23] I. Wald, C. Benthin, and P. Slusallek. Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, pages 77–86, 2003.

[24] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. SCI Institute Technical Report UUSCI-2006-015, University of Utah, 2006. (conditionally accepted at ACM Transactions on Graphics, available at http://www.sci.utah.edu/~wald/Publications/webgen/2006/BVH/download/togbvh.pdf).

[25] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, Sept. 2006.

[26] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. (Proceedings of ACM SIGGRAPH).

[27] S. Woop, G. Marmitt, and P. Slusallek. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Proceedings of Graphics Hardware*, 2006.