

Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes

Maxim Shevtsov, Alexei Soupikov and Alexander Kapustin[†]

Intel Corporation



Figure 1: Dynamic scenes ray traced using parallel fast construction of kd-tree. The scenes were rendered with shadows, 1 reflection (except HAND) and textures at 512x512 resolution on a 2-way Intel [®]Core[™]2 Duo machine.

a) HAND - a static model of a man with a dynamic hand; 47K static and 8K dynamic triangles; 2 lights; 46.5 FPS.

b) GOBLIN - a static model of a hall with a dynamic model of a goblin; 297K static and 153K dynamic triangles; 2 lights; 9.2 FPS.

c) BAR - a static model of bar Carta Blanca with a dynamic model of a man; 239K static and 53K dynamic triangles; 2 lights; 12.6 FPS.

d) OPERA TEAM - a static model of an opera house with a dynamic model of 21 men without instancing; 78K static and 1105K dynamic triangles; 4 lights; 2.0 FPS.

Abstract

We present a highly parallel, linearly scalable technique of kd-tree construction for ray tracing of dynamic geometry. We use conventional kd-tree compatible with the high performing algorithms such as MLRTA or frustum tracing. Proposed technique offers exceptional construction speed maintaining reasonable kd-tree quality for rendering stage. The algorithm builds a kd-tree from scratch each frame, thus prior knowledge of motion/deformation or motion constraints are not required. We achieve nearly real-time performance of 7-12 FPS for models with 200K of dynamic triangles at 1024x1024 resolution with shadows and textures.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism: Ray tracing; Color; Shading; Shadowing and texture I.3.1 [Hardware Architecture]: Parallel processing

1. Introduction

Ray tracing is the core method of photo realistic rendering using global illumination simulation. Excellent ray tracing

[†] {maxim.y.shevtsov,alexey.soupikov,alexander.kapustin}@intel.com

reference is [Gla89]. Since ray tracing requires global access to the entire scene it heavily relies on good acceleration structures to make ray-shooting queries efficient [Hav01].

Kd-tree is the acceleration structure of choice for the majority of current interactive ray tracing algorithms [Hav01, Wal04]. A kd-tree is an axis-aligned BSP tree that splits the scene space using a cost function for the split position. Due to using kd-tree ray tracing algorithm has logarithmic complexity of number of primitives. The cost model for kd-tree construction [MB90] estimates the average cost of traversing an arbitrary ray through the kd-tree, combining a cost of traversal step and a cost of ray-primitive intersection test. A kd-tree construction proceeds in top-down fashion using the estimated cost to determine split plane position in a current node until some termination criteria is reached and the node becomes a leaf.

Although ray tracing of complex static scenes with millions of triangles has become interactive recently [RSH05] fully exploiting parallelism of the rendering stage, dealing with dynamic scenes is an open problem.

Construction of high quality kd-tree is a computationally complex problem difficult to parallelize. Although fast build methods were published [WH06, PGSS06, HSM06, WK06], their performance allows interactive ray-tracing for scenes with a relatively fraction of dynamic triangles (about 100,000). We raise that barrier without implying any restrictions on the dynamic behavior of the scene. The presented method rebuilds kd-tree for the entire scene each frame and does not require any prior information about vertices motion. Its performance also scales linearly with number of threads.

We propose parallel initial spatial partitioning that allows identification and construction of kd-tree branches (sub-trees) by threads independently. This approach takes advantage of the parallelism present to the largest possible degree. Building a sub-tree does not require any synchronization in the inner loop and the sub-tree data can fit in caches of the processor constructing that sub-tree.

2. Previous Work

There is a wide range of hardware architectures suggested to run ray-tracing. Research was done recently for ray-tracing and kd-trees mapped to GPU [Pur04, FS05], and to Cell processor [BWSF06]. Dedicated ray tracing hardware is proposed by [WSS05, WMS06]. We target a shared memory architecture with many CPU-like cores, including recent multi-core CPUs.

A choice of acceleration structure is a critical issue for ray-tracing. The simplicity of using axis-aligned bounding box (AABB) as a proxy for geometry is very attractive for accelerating ray tracing. Nearly real-time performance of acceleration structures directly based on AABBs hierarchy, like BVH [RW80], is obtained for scenes with either a-priori

known motion [WBS07] or with quite small number of moving triangles [LYMT06]. A memory efficient variant was proposed in [WK06]. Benthin in [Ben06] uses AABB for each NURB during construction of the kd-tree to avoid expensive computations. We use triangle AABB as a proxy during kd-tree construction as it has additional advantage for fast SAH estimation since the number of potential split candidates for a given AABB is limited.

Using a uniform grid for dynamic scenes and frustums traversal is presented in [WIK*06]. Even with such extensions a uniform grid has rather poor performance at rendering stage as it lacks an adaptivity to spatial distribution of geometry. Nested grids can better adapt to the scene geometry. Ray traversal algorithm for such structures is not that efficient though, since 3D-DDA algorithm cannot be used. In addition, even with improvements like special coding of empty space grid-based techniques suffer from large storage requirements [Hav01].

Construction of high quality KD-tree is bandwidth hungry and computationally expensive task. Attempts to reduce time spent on kd-tree construction were performed using hybrid data structure combining kd-tree with bounding volumes [HHS06]. Similar combining of BVH and spatial partitioning for increasing overall performance was made in [WK06]. However these approaches still lack parallel implementation and optimized traversal like MLRT [RSH05] and thus demonstrate modest overall performance. Several researchers [WH06] use variations of sweep-and-prune algorithm for kd-tree construction. The sweep requires split candidates at each dimension to be sorted only once at the beginning of the construction. The details of implementation are described in [WH06]. We advocate the approach when no sweep-and-prune algorithm is used to avoid any sorting since its parallel implementation can easily become bandwidth limited for large input models.

The acceleration of kd-tree construction restricting a set of possible splits by space discretization was first introduced in [HKRS02]. Conceptually the same idea was described in [PGSS06]. [WH06] use triangle centroids instead of triangle bounds. Using that approach for a SAH approximation is limited to the case when triangles do not vary much in size. In contrast, we have no restrictions on the primitive size or geometry distribution. We use binning pretty similar to [HKRS02] and [PGSS06]. Several researches [HSM06, PGSS06] increase kd-tree construction speed approximating SAH by a piecewise linear function. However, their implementations do not exploit the full potential of aggressive SAH approximation. The precise SAH evaluation takes about 90% of the construction time in [PGSS06] jeopardizing the benefits of using SAH approximation. We perform the re-binning after selecting a single split rather than trying to look for all splits at the current tree depth. Our approach simplifies memory management and perfectly saves

memory bandwidth since our SAH evaluation algorithm accesses memory in a sequential read-only pattern.

To overcome the algorithmic complexity [PGSS06] introduce a technique of the cost function evaluation at every k -th candidate position but they are still processed at the sorting stage. We propose the technique of skipping all computations for skipped candidates.

One of the bottlenecks of kd-tree construction is memory allocation. The total size of data is growing during construction process. Calling memory allocation functions (malloc, calloc, etc.) is expensive. In a case of parallel implementation the allocation calls add high synchronization overhead. [Ben06] uses a thread-local pre-allocated pool. Its obvious disadvantage is impossibility to estimate the size of the pool in advance. We propose a thread-safe memory pool implementation allocating additional memory so rarely that it doesn't affect performance.

Although some researchers [Ben06, PGSS06] are exploring a possibility of parallel kd-tree construction their results are still far from perfect scalability especially on many threads or cores. Thus [Ben06] present using 2 threads running creation and initial sorting of candidate lists only. Parallel implementation by [PGSS06] has a considerable sequential portion. [IWP07] presents alternative way of handling dynamic scenes. The system performs rendering and re-fitting of current BVH version in multiple threads constructing a new BVH version asynchronously in a single dedicated thread. The system demonstrated impressive performance since in many dynamic cases the quality of re-fitted BVH degrades slowly over multiple frames providing enough time for sequential construction of the new BVH. Handling scenes with fast motion or quickly changing topology is still problematic since in such cases the BVH re-fitting is not usable and full construction is required.

We present a kd-tree construction with all stages running in parallel with minimal synchronization overhead allowing to exploit as many threads as available to achieve fast kd-tree re-build from scratch every frame.

3. Fast construction of kd-trees

Using high-quality kd-tree is essential for achieving interactive ray tracing performance. Therefore, the goal is building a kd-tree as fast as possible minimizing its quality degradation. A typical kd-tree construction proceeds in a top-down fashion by recursively splitting a current node into two sub-nodes using the following sequence of tasks.

- Generate split plane candidates at some locations;
- Evaluate cost function using SAH at each location;
- Pick the optimal candidate (with lowest cost) and perform split into two child nodes;
- Pass over geometry to distribute it among children;
- Repeat recursively;

We follow the similar approach. In this section we focus on the first three stages. We use triangle AABB as a proxy for triangle during fast estimation of SAH. The cost function is piecewise linear thus it needs to be evaluated only at the boundaries of the AABBs that lie inside current node [Ben06]. These locations are also called split candidates.

3.1. SAH-approximation

As shown in [WH06] for a large number of geometric primitives the cost function can be computed in a discretized setting because of its integral form. To overcome an algorithmic complexity we use conceptually similar technique, although our approach works with both large and small objects. Instead of storing object references at each bin we replace a variable size list (or array) with just an object counter. Constructing such a structure requires a single and inexpensive pass over geometry rather than sorting.

3.2. Conventional binning algorithm

Originally binning algorithm (pigeonhole sorting, bucket sorting) algorithm was proposed for points. The idea is to split a 1-D interval into a given number of equally sized bins forming a regular grid. The bin index an object belongs to can be calculated directly from its position. Using a single linear pass over geometry one computes a number of triangles in the bin and updates bin's candidate split value (closest to the bin boundary), see Figure 2. When a triangle represented as a single point a bin where point is located is updated or each bin overlapped by the triangle is updated if the algorithm works with full triangle extent. This data later is used for fast SAH approximation which is very imprecise.



Figure 2: a) Conventional binning algorithms and b) evaluation of SAH using this algorithm

3.3. Min-max binning algorithm

The idea of min-max binning algorithm inspired by [HKRS02] is to keep track of where each triangle AABB begins and ends in two separate sets of bins, see Figure 3. Almost the same technique was described in [PGSS06].

Each bin is just a counter. For each primitive's AABB we update exactly one bin in the first set (where AABB begins) and one bin in the second set (where AABB ends). Thus we completely remove a dependency on the total number of

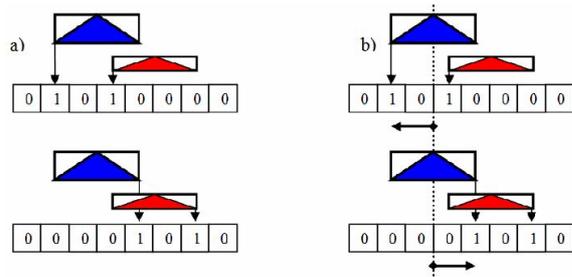


Figure 3: a) min-max binning algorithm and b) evaluation of SAH using this algorithm

bins. This property of the algorithm is essential for *initial clustering* task (section 4.1).

We build min-max bins using separate 1D grid in each dimension. Nevertheless our algorithm is well suited for 2-D and even 3-D binning. The described min-max binning is also well suited for algorithms storing primitive references in bins since it eliminates the need to keep track of references to be unique at leaves. However, we observed that storing individual primitive lists in bins is not necessary since storing counters performs better.

We perform adaptive skipping of the primitives at the higher kd-tree levels. Processing each l -th primitive at the binning step, where $l = \log_{10}(N)$ and N is number of primitives in the current node, allows 3-4x speedup for heavily tessellated objects as N and $\log(N)$ change at each node providing somewhat equal weighting of features. However, even better would be using a low-detailed version of the given scene for the same purpose.

3.4. SAH estimation using min-max bins

SAH approximation algorithm consists of two steps. The first step is a pass over the primitives performing the min-max binning. The second step is a pass over bins estimating SAH values.

Bin boundaries are used as splitting plane candidates. A number of primitives on the left of the split candidate is computed using min-bins set and number of primitives on the right is computed using max-bins set. Min-max binning allows computing exact primitive counters to the left and to the right of a bin boundary disregarding of primitive sizes.

The position minimizing SAH value is selected as a split. An additional pass over the primitives creates arrays of primitive references for the left and the right sub-nodes. We will refer to this pass as a geometry splitting pass. During this pass we also determine the tight boundaries of geometry in the children and adjust the final split plane position to the one of those boundaries if there is an empty space between children.

As other researchers [PGSS06, HSM06] we also switch to exact SAH-evaluation at some tree depth. The important difference we found is that we can use our SAH approximation at much deeper tree levels. Experiments demonstrated that it is beneficial to switch to exact SAH computation when the number of primitives in the current node is less or equal to the number of bins. Experiments show that using the same fixed number of 32 bins at any level is sufficient. *Initial clustering* phase in the parallel algorithm (section 4) requires more though.

3.5. Memory allocation

We implement possibly the fastest way of keeping track of the memory allocation for kd-tree construction. We keep tracking of memory by chunks linked into lists. For each chunk we store a start pointer, a chunk size and an end pointer. The space between begin pointer and end pointer comprises committed memory. Every memory request just shifts the end-pointer by the certain amount of bytes. If there is no memory left in the current chunk to complete request memory manager allocates a new chunk (usually it happens no more than 2-4 times per the whole construction). The manager links chunks in a list such that the last chunk in the list is the last used chunk.

When memory is returned the manager easily finds the corresponding chunk because its boundaries are known (from "begin" to "begin + size of chunk") and shifts the end pointer by the amount of the returned memory. Such memory management requires that the memory for a particular task is returned in the same order as it was committed. Construction of kd-tree in top-down and left-first fashion perfectly fits into such memory management model.

Kd-tree construction uses two types of memory pools. First type is a pool for nodes and leaves of kd-tree that only grows during construction, so it transparently maps to a linked list of chunks. Second type stores primitive indices for the left and right sub-node at each recursive step thus it is a subject to frequent allocation/deallocation. Typically, the kd-tree is built in left-first order which starts with the whole scene and always continuing to construct the left sub-tree, and then right sub-tree, until returning to the root. To handle this we use one pool for left sub-nodes and another pool for right sub-nodes. In this case memory re-allocations always happen at the tail of each pool.

4. Parallel construction of kd-trees

Presented approach is easily extended to the parallel construction of kd-trees in multiple threads. Running task in parallel requires partitioning of the whole task into smaller portions (jobs) assigned to threads.

A straightforward way is exploiting data parallelism at each step of algorithm outlined in section 3. In fact, binning

and geometry splitting passes perfectly run in parallel when each thread is given an equal number of primitives. Memory management is also easy: each thread has its own set of the pools described above. Such approach works well for a large number of primitives. The alternate way is when each thread builds a sub-tree. This requires some sort of initial decomposition of geometry. However, initial decomposition so far was performed sequentially [PGSS06]. We present a parallel solution to this phase too.

The simplest decomposition is even distribution of the primitives among available threads. E.g. each of the 4 threads processes 250K triangles from 1M triangles in scene. Despite of good memory locality, such geometry decomposition has an obvious disadvantage. Kd-trees built by the different threads will overlap in space. There is no known way to merge overlapping kd-trees, while traversing several trees with a ray results in rendering slowdown.

Partitioning of space instead of geometry results in non-overlapping kd-trees easily mergeable into a single tree. A regular space partitioning leads to poor load balancing. Thus processing space regions in parallel requires using geometry distribution information for regions selection.

4.1. Initial clustering algorithm for fast partitioning of the domain

Initial clustering partitions space into regions that are further distributed among threads as jobs. This section describes an efficient clustering that itself runs in parallel.

Initial clustering splits the whole scene AABB into disjoint regions that have roughly the same number of primitives. Clustering is done in the data parallel fashion and then resulting regions are given to each individual thread to build local kd-trees (see Figure 4). Initial clustering starts with parallel binning ("binning" stage at Figure 4). Each thread runs min-max binning algorithm processing its own set of primitives. Multiple pairs of thread-local bins are merged into a global bins pair via counters summing.

Using the global binning data we quickly estimate an approximate object median position. The whole spatial domain is split into two sub-domains by the boundary of a bin corresponding to the median value. We repeat median split step using the same binning data in each sub-domain until the number of sub-domains is equal to the number of threads (or a multiple of the number of threads for a finer granularity). The second pass over primitives creating arrays of primitives for each sub-domain is also parallel ("assignment" stage at Figure 4). It requires only $O(\log_2(T))$ decomposition tree levels, where T is the number of threads, see Figure 5. The sub-domains are disjoint and contain approximately equal number of the primitives. For scenes with roughly uniform triangle densities (like the majority of heavily tessellated or scanned models) building sub-trees in individual threads is naturally load-balanced.

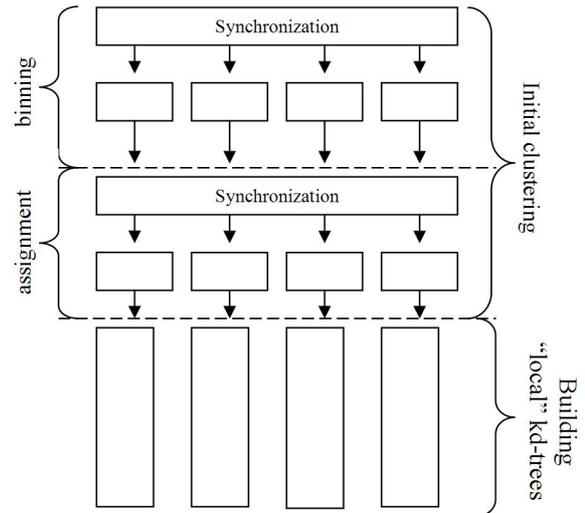


Figure 4: Hybrid parallelization scheme which does parallel initial decomposition (clustering) of data to create jobs for independent processing.

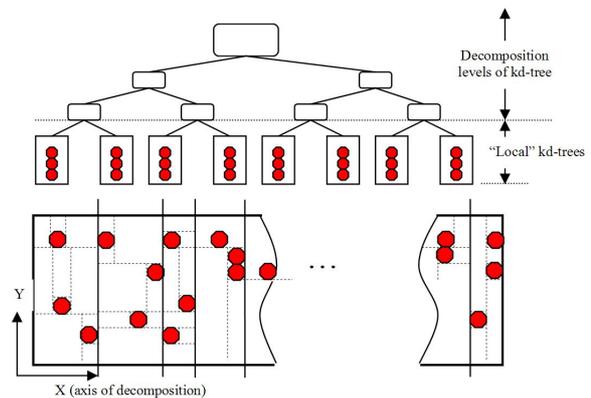


Figure 5: Balanced decomposition by initial clustering

In general case though resulting sub-domains might require building trees of different complexity (i.e. depth) leading to different per thread construction time. To balance the load we break the construction of a sub-tree into smaller tasks inserted and fetched by the threads into/from a shared task pool dynamically. Currently we use the simple and straightforward method of tasks creation. When constructing both children a thread inserts the right child into the pool and proceeds with construction of the left child. The tasks are generated at the upper levels of the thread-local subtree to keep the number of tasks reasonable and avoid a synchronization overhead by limiting the task pool size. We experimented with different sizes of the task pool (up to 200K tasks) and found that for 4 threads 256 task entries would be optimal.

5. Implementation

Bulk of the construction stages use primitive AABBs instead of the actual primitives. AABBs are stored in large arrays. Since construction processes each axis separately we store AABBs as structure of arrays of bounds (SOA) rather than array of structures (AOS).

Number of bins is the most important parameter of SAH approximation based construction since kd-tree quality depends on it. As in [PGSS06] we perform re-binning at each level of recursion which leads to a very good adaptive refinement of bins resolution. Re-binning with a small constant number of bins results in exponential growth of binning resolution with tree depth, so it always outperforms re-using a single set of high resolution bins constructing a large number of tree levels. E.g. re-binning with 32 bins at depth level 3 is approximately equivalent to 256 ($32 \cdot 2^3$) at the top level.

Number of bins for initial clustering stage should be high since we build clusters from a single set of bins without re-binning. In our experiments, we observed that $512 \cdot T$ bins, where T is the number of construction threads, is sufficient. Current implementation performs initial clustering over a single selected dimension. Using 2D min-max bins (2D grids of counters of AABBs min and max vertices) for initial clustering produces higher quality kd-trees for many scenes as it provides a choice of 2 dimensions for split selection maintaining reasonable object median approximation when bins are re-used. So this is the topic for further investigation. On the other hand even for 128 threads only $7 = \log_2(128)$ tree levels are generated by initial clustering when typical tree depth range is 30-50 for moderately complex scenes, so the cost of switching to 2D or 3D bins might consume all the benefits. The second reason is that accurate SAH estimation is much more important for deep tree levels.

We implemented our parallelization approach around the existing sequential kd-tree construction algorithm fully reusing the existing serial code. Parallel initial clustering (together with AABB, normal computations and conversion to accelerated format for each triangle) takes about 10% of the total kd-tree construction time. Parallel construction of subtrees takes the rest 90% of the time.

Usually a dynamic scene has some static environment. Since the offline kd-tree builder for static scenes produces much higher quality kd-tree, handling static and dynamic geometry in two separate kd-trees often results in best performance. Especially in cases when large fraction of rays hit static geometry (like game characters in a building).

Primitives are often combined into high-level geometric objects that are further combined into a scene graph hierarchy. Using this global information to build an individual kd-tree for each object and building a top-level kd-tree treating objects as primitives works really well in case of large number of non-deforming objects. [Wal04] demonstrated impressive performance of such approach for the case of

multiple instances of the same geometry. Replacing initial clusters built by the brute force pass over the primitives (like object median based presented in this paper) with the clusters provided by the global scene information (like nodes of the scene graph) for dynamic scenes produces ambiguous results.

Since our kd-tree building algorithm is not restricted to a certain primitive type, it easily handles other kd-trees as primitives. "OPERA TEAM" scene on Figure 1 consists of small static portion and large number of individual dancers that can be handled as lower-level kd-trees.

We performed some number of experiments with constructing the top-level kd-tree over the set of kd-trees. We found that construction of top-level kd-tree using global information pays off well for 4 and more different non-overlapping objects (or for heavily occluded scenes). However, in case when different kd-trees overlap the top-level kd-tree performance degrades heavily.

6. Results

We have tested our interactive ray-tracing on a 2-way Intel®Core™2 Duo machine (so 4 threads on 4 cores). We employ multi-threaded 4x4 SIMD ray packet traversal. We intentionally don't perform any tweaking of building parameters for any particular scene or any particular type of motion.

To evaluate efficiency of our technique we compare rendering speeds for the same scene using kd-trees produced by the off-line high quality building algorithm and by dynamic builder presented here, see Table 1. With rendering speed around 70% of the speed measured for highly-optimized trees our approach allows 120-300x construction time speed up.

| scene and # triangles | Highly optimized kd-trees | | Our routine | |
|---|---------------------------|---------------|---------------|-----------------|
| | constr. time | render. perf. | constr. time | render. perf. |
|  Happy Buddha 1087K | 90.1 s | 22.1 fps | 0.45 s | 15.4 fps |
|  Blade 1765K | 88.2 s | 11.04 fps | 0.69 s | 7.8 fps |
|  Soda Hall 2195K | 152.7 s | 25.8 fps | 0.47 s | 19.3 fps |

Table 1: Construction time and rendering performance. Rendering performance numbers are for 1024x1024 resolution, using lighting (1 point light source) and shadows.

Memory consumption becomes the important factor when constructing a kd-tree for a large model. We measured the peak size of memory footprint for models with 7M and 10M triangles, see Table 2. Analysis shown that full-featured off-line builder we used as reference in the previous table consumes over 2GB for each case. So with respect to memory consumption our algorithm is promising for up to 10M models.

| scene and # triangles | constr. time | render. perf. | peak memory |
|---|--------------|---------------|-------------|
|  Asian Dragon, 7219K | 1.7 s | 2.9 fps | 1.24 Gb |
|  Thai Statue, 10M | 2.46 s | 3.14 fps | 1.42 Gb |

Table 2: Construction time, rendering performance and memory footprint size for large models. Performance numbers are for 1024x1024 resolution, using lighting (1 point light source) and shadows.

We compare time to image that includes tree construction, ray tracing, and simple shading with data published for the fastest rendering systems, see Table 3.

Fast kd-tree construction algorithms for dynamic scenes produce lower quality kd-trees comparing to off-line builders for static scenes. So time to image for dynamic scenes depends on both resolution and kd-tree construction time, see Figure 6. Higher resolutions require better kd-tree quality than lower resolutions. Automatic kd-tree parameters tuning depending on given resolution is a subject for further research.

We also compared our dynamic scene performance to the recent publications using the same 3D data. We construct the kd-tree from scratch every frame without any prior knowledge of motion. "Cowboys" scene demonstrates the idea of handling static and dynamic geometry in two separate kd-trees (see section 5). Table 4 shows comparison results taking in account total construction and rendering time. Table 4 shows that we achieve interactive frame rates for high resolution on scenes of various complexity and structure. To address difference in measurements due to different resolutions, CPU speeds and number of cores in table 4 we report our results for one and 4 threads (using different CPUs and resolutions).

To make sure that the initial clustering of geometry is optimal for load balancing and does not affect kd-tree quality we constructed kd-trees using variable number of threads and measured rendering speed with fixed number of threads. As rendering speed reduces only slightly (10-15%) we can conclude that degradation of kd-tree quality with the increasing

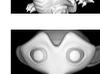
| scene and # triangles | [WH06] | [WK06] | Our routine | Our routine |
|---|--------------------------|--------------------------|--------------------------|---------------------------|
| | AMD Opteron | Intel P4HT | dual-Intel Core2 Duo | dual-Intel Core2 Duo |
| | 2.6 GHz | 2.8 GHz | 3.0 GHz | 3.0 GHz |
| | 640x480 1 core | 640x480 1 core | 640x480 1 core | 640x480 4 cores |
|  Shirley Scene 6, 804 | n.a. | 0.083 | 0.023 | 0.006 |
|  Happy Buddha, 1087K | 32.2 | 1.837 | 0.696 | 0.181 |
|  Stanford Bunny, 69k | 4.8 | 0.176 | 0.104 | 0.027 |
|  Stanford Dragon, 863k | 23.9 | 1.557 | 0.751 | 0.195 |
|  Blender Suzanne, 251K | n.a. | 0.448 | 0.235 | 0.062 |
|  Ward Conf., 1065K | n.a. | 1.523 | 0.628 | 0.161 |

Table 3: Time to image (in seconds) for [WH06] and [WK06], data were taken from [WK06], in comparison with our approach.

number of construction threads is negligible. Figure 7 shows rendering speed for 4 threads with kd-tree constructed by 2 to 256 threads.

We tested scalability of presented technique on SMP machine with 4 cores, see Figure 8. It shows that FPS count for both construction and rendering grows roughly linearly with number of threads achieving 3.9x speedup for 4 threads. Scalability is slightly better for large models.

7. Conclusion and Future Work

We propose new parallel linearly scalable algorithm for fast construction of acceleration data structure at each frame without prior knowledge about geometry motion. Resulting kd-tree is fully compatible with highly optimized traversal using frustum implementation like MLRT [RSH05].

We demonstrated that conventional kd-tree is suitable data structure for interactive ray tracing of dynamic scenes as we presented the fast construction algorithm producing kd-tree of reasonable quality. The algorithm benefits from us-

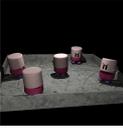
| scene, # triangles and # lights | BVH AMD Opteron 2.6 GHz 1024 x 1024 1 core | BIH Intel P4HT 2.8 GHz 640 x 480 1 core | Our routine Intel P4HT 2.8 GHz 1024 x 1024 1 core | Our routine Intel P4HT 2.8 GHz 640 x 480 1 core | Our routine dual Intel Core2 Duo 3.0 GHz 640 x 480 1 core | Our routine dual Intel Core2 Duo 3.0 GHz 1024 x 1024 4 cores |
|---|--|---|---|---|---|--|
|  Toys, 11K animated triangles, 1point light | 10.5 fps* | n.a. | 3.0 fps | 9.0 fps | 22.9 fps | 23.5 fps |
|  FairyForest, 178K animated triangles, 2point lights | 2.16 fps* | 1.79 fps | 0.8 fps | 2.0 fps | 3.1 fps | 5.84 fps |
|  Cowboys, 63K static and 171K animated triangles, 2point lights | n.a. | n.a. | 1.0 fps | 2.3 fps | 5.0 fps | 7.2 fps |

Table 4: Framerate comparison results for 3 dynamic scenes. BVH times from [WBS07], BIH times from [WK06].

*Note that [WBS07] perform the analysis of all existing poses (e.g. it takes 31 seconds for Fairy scene with only 20 key frames) of the models before animation.

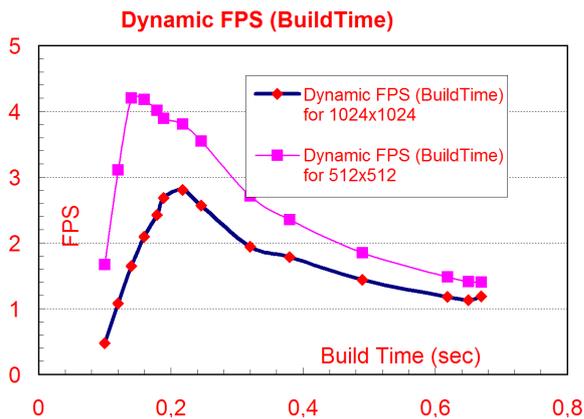


Figure 6: Build time vs. rendering time in terms of resulting FPS for different resolutions. The rendered dynamic scene is the same.

ing multiple construction threads. Presented memory pools management is inherently thread-safe and efficient.

Described technique allows for ray tracing of complex animated models at a performance that is better than any ray tracing performance for dynamic models we are aware of.

In future we plan investigation in the following directions. Since kd-tree construction requires multiple passes over geometrical primitives it is a bandwidth hungry task. So careful memory bandwidth handling is a top priority. To address that

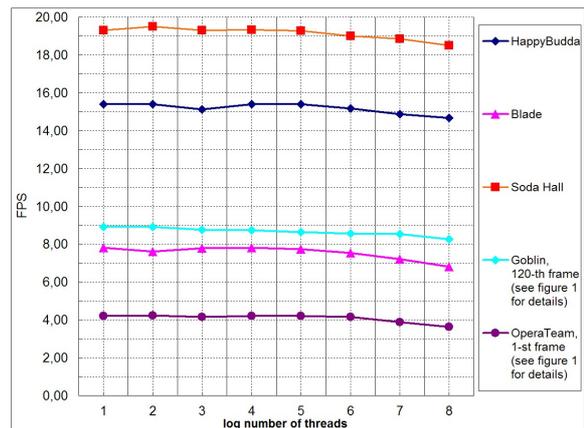


Figure 7: Quality of kd-tree with number of threads that take part in construction. Rendering performed at 1024x1024 resolution, using lighting and shadows (Goblin - 2 point lights, OperaTeam - 4 point lights, the other models - 1 point light).

we can reduce the work set for passes over the primitives (e.g. compress AABB data using quantization), reduce the number of passes over large number of primitives (e.g. reusing the binning information) and offload some of the kd-tree construction steps to a GPU. Vectorization using SIMD instructions will increase performance of the construction algorithm.

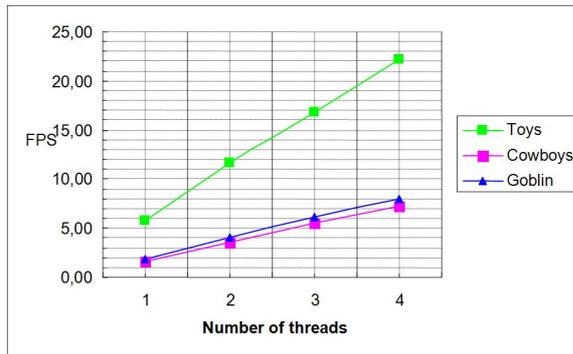


Figure 8: Performance scaling with number of CPUs. Performance numbers are for 1024x1024 resolution, with lighting and texturing. The details of the Toys and Cowboys scenes could be taken from Table 4, Goblin scene is described at caption of Figure 1.

The algorithm presented in the paper is perfectly suitable for on-demand (aka lazy) construction of kd-tree nodes so we plan to experiment with such implementation.

Since quality of kd-tree constructed by offline builder is approximately 1.43x better we plan to research benefits of enabling various cost reduction methods depending on a tree level. For example, using the cost model at initial clustering stage (using 2D or 3D binning), bins re-using at high levels, re-binning at lower levels, using split plane - triangle intersection points as split candidates (aka perfect splits) at very low levels when the working set fits into caches.

Large number of threads and complex construction techniques will require more careful management of tasks at a finer granularity, because using the cost model can produce tasks with number of primitives different by order of magnitude.

Automatic calculation of kd-tree construction parameters for different image resolutions is also a future research subject.

References

[Ben06] BENTHIN C.: *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University, 2006.

[BWSF06] BENTHIN C., WALD I., SCHERBAUM M., FRIEDRICH H.: Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 25–23.

[FS05] FOLEY T., SUGERMAN J.: KD-tree Acceleration Structures for a GPU Raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS confer-*

ence on Graphics hardware (2005), ACM Press, pp. 15–22.

[Gla89] GLASSNER A.: *An Introduction to Raytracing*. Academic Press, 1989.

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague, 2001.

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

[HKRS02] HURLEY J., KAPUSTIN A., RESHETOV A., SOUPIKOV A.: Fast Ray Tracing for Modern General Purpose CPU. In *Proceedings of the GraphiCon 2002* (2002).

[HSM06] HUNT W., STOLL G., MARK W.: Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), IEEE Computer Society.

[IWP07] IZE T., WALD I., PARKER S. G.: Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization* (2007).

[LYMT06] LAUTERBACH C., YOON S.-E., MANOCHA D., TUFT D.: RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 39–46.

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *The Visual Computer* 6, 3 (1990), 153–166.

[PGSS06] POPOV S., GÜNTHER J., SEIDEL H., SLUSALLEK P.: Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[Pur04] PURCELL T. J.: *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-Level Ray Tracing Algorithm. *ACM Transactions of Graphics* 24, 3 (July 2005), 1176–1185. Proceedings of ACM SIGGRAPH 2005.

[RW80] RUBIN S., WHITTED T.: A 3d representation for fast rendering of complex scenes. In *Proceedings of the ACM SIGGRAPH 1980* (1980), pp. 110–116.

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics* 26, 1 (2007).

- [WH06] WALD I., HAVRAN V.: 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* (2006), pp. 61–69.
- [WIK*06] WALD I., IZE T., KENSLE A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions of Graphics* 25, 3 (July 2006), 485–493.
- [WK06] WÄCHTER C., KELLER A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006 (Proceedings of 17th Eurographics Symposium on Rendering)* (2006), Akenine-Möller T., Heidrich W., (Eds.), pp. 139–149.
- [WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware* (2006), pp. 67–77.
- [WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions of Graphics* 24, 3 (July 2005), 434–444. Proceedings of ACM SIGGRAPH 2005.