Fast, Parallel, and Asynchronous Construction of BVHs for Ray Tracing Animated Scenes

Ingo Wald [a,b] Thiago Ize [a,b] Steven G. Parker [b]

[a]*Advanced Graphics Lab, Intel Corporate Technology Group, Santa Clara, CA, 95054*

[b]*SCI Institute, University of Utah, 50S Central Campus Dr., Salt Lake City, UT, 84112*

Abstract

Recent developments have produced several techniques for interactive ray tracing of dynamic scenes. In particular, bounding volume hierarchies (BVHs) are efficient acceleration structures that handle complex triangle distributions and can accommodate deformable scenes by updating (refitting) the bounding primitive without restructuring the entire tree. Unfortunately, updating only the bounding primitive can result in a degradation of the quality of the BVH, and in some scenes will result in a dramatic deterioration of rendering performance. In this paper, we present three different orthogonal techniques to avoid that deterioration: (a) quickly rebuilding the BVH using a fast, binning-based approach; (b) a parallel variant of that build to better exploit the multi-core architecture of modern CPUs; and (c), asynchronously rebuilding the BVH concurrently with rendering and animation, allowing it to scale to even larger models by stretching the (parallel) BVH build over one or more frames. Our approach is particularly targeted towards future "many-core" architectures, and allows for flexibly allocating how many cores are used for rebuilding vs. how many are used for rendering.

1 Introduction

In the last decade, the graphics community has benefited from tremendous improvements in the performance and capabilities of PC based graphics cards, with GPUs now providing around 330 GFlops and near-CPU like programmability [4]. This demand for faster and more programmable GPUs is driven mainly by the demanding needs of video games for faster and more realistic graphics.

Along with the tremendous improvements in GPUs, CPUs are also becoming much faster, especially with the current trend of increasing the number of cores per chip. For instance, a commodity 3 GHz quad-core Clovertown today has roughly 96 GFlops, a PlayStation 3's CELL processor has 180 GFlops, and Intel has already announced the "tera era" in which "many-core" architectures will be commonplace.

Email addresses: `wald@sci.utah.edu` (Ingo Wald), `sparker@cs.utah.edu` (Steven G. Parker).

The quest for increased quality, combined with increases in available compute power has led to a stunning pace of development for rasterization-based GPUs. In addition, it has also reignited an interest in ray tracing. Ray tracing can easily fulfill the growing quality demands, such as soft shadows, depth-of-field, caustics, participating media, and global illumination; but the main limitation is that it is not yet efficient enough for use in applications such as games, which demand real-time performance, high quality, and highly dynamic geometry.

Notwithstanding this lack of performance on today's hardware, as long as compute power continues to rise ray tracing will eventually become real-time. With this in mind, many researchers have recently focused on realizing real-time ray tracing, and, more recently, on ray tracing dynamic scenes (see, e.g., [25] for an overview). Today, real-time ray tracing with dynamic scenes can be realized via either kd-trees, grids, or bounding volume hierarchies (BVHs), but there are trade-offs associated with each of these data structures [25]. Kd-trees seem to offer the highest ray tracing performance, but are most costly to build [23]; grids are efficient to build, but rely on a high degree of ray coherence which may not exist for complex scenes and/or secondary rays [24]. BVHs offer a compromise between performance and the ability to handle complex scenes and secondary rays, but are currently limited for many types of dynamic scenes. In particular, BVH-based interactive ray tracing systems are currently optimized for scenes that deform over time, and will deteriorate in performance for unstructured motion or severe deformations [22]. While one can rebuild a deteriorated BVH every few frames to restore performance, this creates a disruptive pause while the BVH is being rebuilt [11].

In this paper, we propose a new approach for handling dynamic scenes in a BVH-based ray tracer that is particularly targeted towards the highly parallel architectures we expect ray tracers to run on in the near future. In particular, we present three different, orthogonal techniques to avoid that deterioration: (a) quickly rebuilding the BVH using a fast, binning-based approach similar to recent scan-based kd-tree build methods; (b) a parallel variant of that build to better exploit multi-core architectures; and (c), an approach for asynchronously rebuilding the BVH concurrently with rendering and animation, allowing scalability to even larger models by allowing the (parallel) BVH build to run over multiple frames if required [1] :. These three techniques allow for (a) flexibly balancing the number of cores allocated to rendering and rebuilding; (b) quickly rebuilding from scratch if required; and (c) rendering scenes faster than could be done by from-scratch rebuilding alone, especially for larger scenes.

2 Background

Real-Time Ray Tracing and Dynamic Scenes. As early as the 1990s, researchers achieved interactive ray tracing performance using massively parallel supercom-

[1] Essentially, we are combining three techniques that have previously appeared in part in [8] and [20], and are described in more detail in these original works.

puters [3, 13, 14]. With the growing capabilities of commodity architectures, such compute power can now be found even in commodity GPUs and multi-core CPUs. In particular, since Reshetov's "Multilevel Ray Traversal" [16], PC based ray tracers are—at least for simple shading—able to achieve fully interactive frame rates for non-trivial scenes on multi-core desktop PCs. Since then, near real-time performance has been demonstrated on a variety on architectures, and with a variety of data structures and traversal algorithms [25].

Apart from low level optimizations, fast ray tracing depends on using efficient spatial acceleration structures, such as a BVH [17], kd-tree [10], or grid [1]. While there has been a long-running debate on which of these is best, by 2005 virtually all fast ray tracers were built on kd-trees. Unfortunately, kd-trees are costly to build and cannot easily be incrementally updated, and as such present an obstacle to handling dynamic scenes. Thus, researchers have again started to actively explore better ways to support dynamic scenes with other acceleration structures [25].

As a result of these efforts, ray tracing animated scenes has been demonstrated using both kd-trees [2, 7, 15, 18], grids [24], and BVHs [11, 22, 27], with various performance-vs-flexibility trade-offs depending on which data structure is being used. Generally speaking, grids are currently considered to be fastest to build but somewhat less efficient for traversal/intersection; kd-trees to be the most efficient for traversal/intersection but most costly to build, and BVHs somewhere in-between in build time, and close to kd-trees for traversal.

Surface Area Heuristic. In addition to the data structure being used, a ray tracer for dynamic scenes has to decide on how much effort it invests into the quality of the data structure. In particular for hierarchical data structures, one can use either a fast spatial median split, or a more sophisticated, cost estimator-driven build algorithm. Most kd-tree and BVH based ray tracers today employ a *surface area heuristic* (SAH) for building the data structure. Given a spatial region V with N primitives, the SAH provides an effective tool for estimating the expected cost of any partitioning of (N,V) into two halves (N_L,V_L) and (N_R,V_R). Using several more or less reasonable assumptions (see [23] and [5] for more details), the SAH estimates the traversal cost of such a partition as

$$C_{(N,V)\to(N_L,V_L)+(N_R,V_R)} = K_T + K_I\left(\frac{SA(V_L)}{SA(V)}N_L + \frac{SA(V_R)}{SA(V)}N_R\right),$$

where $SA(V)$ is the surface area of V, and K_T and K_I are some implementation-specific constants. Using this cost estimator, a greedy SAH build tests all possible partitions, greedily selects the one with least expected cost, and recurses.

BVH-based Dynamic Scene Ray Tracing. As mentioned above, ray tracing animated scenes today is possible for each of the three dominant acceleration data structures—kd-trees, grids, and BVHs. Among those, BVH has long been (wrongly) suspected of yielding inferior traversal performance, but had always been recognized to be well suited for deformable geometry. Instead of subdividing space into "voxels" of triangles, BVHs build an *object hierarchy*, and in each tree node store a bounding volume for that subtree's geometry. Thus, a BVH is defined through two parts: the tree topology, and each tree node's bounding volume. Once the objects move, instead of rebuilding the complete BVH from scratch, one can also leave the topology unchanged and only *refit* the BVH nodes' bounding volumes. While this refitted BVH may have a different and potentially less efficient tree structure than one built from scratch, the refitted BVH will nonetheless be correct. Refitting a BVH is extremely fast, and often less costly than the associated animation updates.

With that in mind, several researchers have recently proposed refitting-based ray tracers, some of which achieve performance close to Reshetov's MLRT system [22].

Handling BVH Deterioration. While refitting a BVH is inexpensive, it does have several drawbacks. First, it is only applicable for *deformable* scenes (i.e., scenes that do not change the triangle count or vertex connectivity). Second, refitting a BVH will result in a *correct* BVH, but it will not necessarily be *efficient*. The refitted BVH retains the original frame's BVH topology, but as the scene deforms the triangles might form a configuration for where a different structure might yield better performance. This will eventually lead to a deterioration of BVH quality (and performance) as scene and BVH become out of sync.

As pointed out by Lauterbach et al. [11], deforming a BVH usually works for at least some number of frames, and instead of rebuilding a BVH every frame, one could rebuild only every few frames, with the frames in-between handled by BVH deformations. In order to do these rebuilds when they are most effective, Lauterbach et al. [11] have proposed a "rebuild heuristic" that detects BVH degradation, and rebuilds the BVH if and only if the quality degradation has reached a given threshold. This allows for striking a balance between total rebuild cost and render cost, and can yield a significantly reduced average frame time in an animation.

Unfortunately, a lower *average* frame time is not always helpful in an interactive setting. In an offline animation the infrequent rebuilds can be amortized over all frames of the animation, yielding a low average time per frame; in an interactive setting, however, amortization does not apply, and system responsiveness is disrupted while a rebuild is performed, which hurts the user's ability to interact with the environment.

In order to distribute the rebuild cost more evenly over multiple frames, Yoon et al. [27] have proposed to selectively restructure an existing BVH on a frame by frame basis—thereby incrementally fixing some of the accumulated deterioration. For every frame, they first deform the existing BVH, then analyze this BVH to detect nodes of high overlap (i.e., deterioration), and selectively restructure pairs of nodes to reduce this overlap. While very effective at avoiding deterioration, the method also carries a certain cost, may be non-trivial to parallelize, and has not yet been demonstrated in a real-time setting.

As yet another alternative, we recently proposed an approach in which BVH building is performed asynchronously to rendering [8], and to bridge the latency until a new BVH is built by relying on refitting in the mean time. Doing so allows for rendering at a rate that is several times higher than the rate of rebuilding, and thus decouples frame rate from rebuild performance. However, if rebuilding takes too long, performance can begin to deteriorate as the deformation increases. Furthermore, by the time the new BVH is ready, that BVH will already be outdated and no longer optimal, all of which results in a lower and inconsistent frame rate, which forced this system to use a faster—but less efficient—median-split BVH. Since we essentially propose an extension of that technique, we will later on describe it in more detail.

Fast Construction of SAH hierarchies. Instead of only partially or infrequently rebuilding a BVH, the simplest and most robust method for handling dynamic motion would be to rebuild every frame. This is in fact the biggest advantage of grid data structures, which can be easily rebuilt per frame [9].

For kd-trees, fast "scan-based" approaches to building SAH based data structures have been proposed independently by Popov et al. [15] and Hunt et al. [7]. Though these two publications report rebuild rates of roughly 150,000 and 300,000 triangles per second, respectively, this is still insufficient for achieving truly interactive frame rates for anything but rather small models. More recently, Shevtsov et al. [18] have shown that these techniques can also parallelized in a scalable fashion, and that near-real time rates can be achieved on multi-core processors. If some information from the scene hierarchy is available, lazy building from the hierarchy can be up to an order of magnitude faster than building from a triangle soup [6].

For BVHs, fast rebuilding has received far less attention. The fastest known (single-threaded) build method is the BVH variant of the fast spatial median build Wächter et al. [19] have proposed for their bounding interval hierarchy (BIH). Though subtly different from a standard median build [25], this BIH build essentially performs spatial median splits, and thus achieves lower traversal performance than SAH-based builders.

More recently, in a related paper [20] we have shown that fast scan-based kd-tree build techniques can also be applied to BVHs, and that they work at least as well for BVHs as they do for kd-trees. Using these techniques on a dual 2.6 GHz Clovertown PC, single-threaded rebuild performance of up to 1–2 million triangles per second have been achieved. Using parallelization, this rate could be raised to up to 7 million triangles per frame, at which rate the system on that particular hardware stopped scaling, apparently due to bandwidth limitations. Since we use that algorithm in our system, we will describe it in more detail below.

3 System Overview

Summarizing previous work, we can conclude that BVHs hold promise for ray tracing dynamic scenes, but that each of the individual techniques proposed for handling animations has limitations: refitting is fastest, but eventually leads to degraded performance if deformations become too severe; infrequent rebuilding amortizes rebuild cost but leads to disruptions in frame rate; asynchronous building avoids these disruptions, but is limited by how fast any single thread can build the BVH, and can still suffer from severe deterioration if rebuilding takes too long; fast builds reduce the build time but are not fast enough on their own; and finally, parallel binning is even faster, but does not scale once the bandwidth wall is hit, and still limits frame rate to how fast a full BVH can be built.

These problems will likely become exacerbated by future ray tracing systems running on architectures with a large number of cores. However, by combining some of these techniques in the right way, we can cancel their respective disadvantages. In particular, we can combine the fast and parallel BVH building [20] with asynchronous rebuilding [8], which leads to three distinct advantages:

- replacing the asynchronous system's single-threaded builder with a fast, parallel builder significantly reduces the time the asynchronous builder has to wait for a new BVH, and thus reduces the potential for deterioration. At the same time, it also provides the system with an SAH-based BVH instead of a BIH-style builder.
- For scenes where rebuilding is still too slow for per-frame rebuilds, the asynchronous approach allows for amortizing the build time over multiple frames.
- Even though the parallel builder does not scale to all CPUs, building asynchronously allows for using all of the system's cores by assigning as many cores as reasonable to rebuilding, and all other ones to rendering.

4 Fast Construction of SAH based BVHs

In any ray tracer, there is a trade-off between build quality and build time. Better data structures achieve higher performance, but if building them takes too long, the total frame time may still go up. While the asynchronous approach somewhat decouples build time from render performance, the basic problem still exists: If the build takes too long, too much deterioration can occur before a new BVH is built, resulting in an slower and non-smooth frame rate.

Consequently, our original paper on building the BVH asynchronously [8] recommended a spatial median BVH instead of a better SAH BVH. Though yielding somewhat better BVHs, the high-quality SAH build took about $20\times$ longer to build, which eventually led to too much deterioration. Since then, however, Wald [20] has shown that high-quality BVHs can also be built much faster when using the same techniques as recently proposed for fast kd-tree construction [7, 15]. Since these techniques are nearly as fast as spatial median splits and nearly as good as SAH builds, we have decided to adopt them in our system.

In particular, these techniques do not evaluate all possible kd-tree split planes, but use a given set of sample planes. The triangles are then projected into the "bins" formed by these planes, and the number of triangles in each bin are recorded. The algorithm then evaluates the SAH for each plane, and selects the one with lowest cost. Though approximate in nature, for kd-trees it has been shown that as few as 8 planes usually suffice to be within a few percent of the optimum [7].

Though originally designed for kd-trees, the same techniques apply to BVHs as well. The only difference is that in the kd-tree case triangles overlapping the plane have to be counted in both halves, and the plane determines the exact bounds of each subtree. In a BVH, a triangle can be on only one side of the plane, and consequently, the subtree bounds can overlap the split plane. Therefore, we track not only the number of triangles in each bin, but also the bounds of all triangles projecting to this bin. Based on these bins, we can then compute the number of triangles as well as their spatial extent to the left and right of each sample plane, compute the SAH, and select the best plane.

During the build, we never need a triangle's exact shape, but only its axis-aligned bounding box (AABB), as well as one representative point for binning (for which we choose the AABB's centroid). To make use of SSE SIMD extensions, we store centroids and AABBs in SSE 4-float format, and precompute those in the beginning. During the build, each binning operation is then only a few instructions long.

As investigated in more detail in [20], using this method we can build a SAH based BVH roughly 10$\times$ faster than the sweep based SAH build outlined in [22]. Though approximate, high-quality BVHs can be built with as few as 4 bins, and 16 bins are usually close to the optimum (see Table 1). At 16 bins, the near-optimal SAH build is only about 2$\times$ slower to build than a spatial median build, while still producing high-quality BVHs.

scene	Fairy Forest 2	BART	Expl. Dragon
sweep	3006ms (100%)	2270ms (100%)	1614ms (100%)
BIH	216ms (72%)	104ms (75%)	70ms (87%)
binned	364ms (94%)	347ms (95%)	185ms (104%)

Table 1
Single thread absolute build times and relative render performance to SAH sweep for an SAH sweep build, median-split BIH-style build, and fast binned SAH build, for the animated scenes described in Section 7. Build times are in milliseconds, on a 3.0 GHz Opteron CPU (one thread).

5 Parallel Construction of SAH based BVHs

There has been a recent explosion in the number of processor cores available on consumer computers, with high-end PCs already containing 8 or more CPU cores, and a G80-GPU essentially having 16 32-wide SIMD cores on a single chip[2]. We argue that this trend is likely to continue and that soon, architectures with dozens of cores will be commonplace. With that in mind, designing algorithms in a parallel way will soon be critical, in particular for real-time applications like ray tracing.

The obvious way of parallelizing an algorithm producing a binary tree is to have different threads work on different subtrees. Though simple and effective, this technique requires a certain number of subtrees to work on. One way of doing that is to start with a single thread working on the root node, and adding another thread after every split. While simple to implement, not all threads are active from the beginning. And since the first splits unfortunately are the most costly ones, this

[2] While a G80 is often pictured as having 128 scalar processors, these are actually grouped into 16 ”multiprocessors” that each run a warp of 32 threads in SIMD fashion.

technique does not scale well. We therefore propose a two-stage approach: in the first stage, all threads collaboratively generate a coarse partitioning; the second stage then switches to different threads that work on different subtrees [3] .

Setup. Before doing anything else, we first have to compute the required bounding boxes and centroids. Parallelizing this section fortunately is trivial, with each of the T thread computing boxes and centroids for one T^{th} of the triangles.

Stage 1 – Coarse Scene Partitioning. Two alternative strategies can be applied:

GRID-BASED SPLITTING. We take the bounding box of the triangles' bounding box centroids, and uniformly subdivide it into a grid of $k_x \times k_y \times k_z$ cells. Each of the T threads create their own copy of that grid, then takes one T^{th} of the triangles, and in a single pass over those triangles bins each triangle into the cell that its centroid projects to. Once all threads are done, the T grids are merged in parallel (similar to Ize et al.'s parallel grid build [9]). Once merged, one of the threads builds the BVH nodes that represent that grid, which is very similar to Wächter's BIH technique.

PARALLEL BINNING. Alternatively, we can also have the T threads collaborate on each individual partition. First, each of the T threads computes a bin histogram as described in the previous section, albeit only for one T^{th} of the triangles. Once done, one thread merges these histograms, selects the best split, and creates a new BVH node. Based on the selected split, each of the threads then splits its part of the triangles into a left and right half, and copies the two halves into the respective regions of the triangle ID array (which fortunately can be predetermined by the known sizes of all threads' bins).

Stage 2 – Parallel subtree building. Once the coarse scene partitioning is finished—either via parallel binning or via grid-based partitioning—we can process the individual subtrees in parallel. To obtain good load balancing we first sort the subtrees by descending size. Then, the render threads dynamically request subtrees to work on using an atomic read-and-increment on a shared counter, until all subtrees are finished. Since we know that a subtree of N triangles can use at most $2N-1$ nodes, we can simply assign each subtree to one contiguous region of the node array. Thus, we do not have to perform any costly memory allocations, and the only synchronization primitive in that stage is the atomic increment mentioned above.

Though originally designed for parallel SAH construction, the same framework can also be used for parallel spatial-median building. In particular, the grid-based splitting essentially constructs a spatial median build in the coarse partitioning, anyway, so all that needs to be done to have a parallel spatial median BIH builder is to use the existing single-threaded BIH builder in the parallel subtree phase. This way, we eventually have three different parallel build schemes: parallel SAH binning on the top with a binning for each subtree results in a SAH throughout the tree, grid partitioning with binning for the subtrees results in a spatial median at the top and SAH for each subtree, and grid partitioning with a BIH build per subtree results in a spatial median throughout.

[3] This technique has been described in part in [20].

Results. In Table 2, we give build performance and relative build performance for these two parallel build methods. Due to space considerations, we only report data for 2, 4, and 8 threads, as on the particular hardware we use (an 8-way dual-processor Opteron) the parallel build stops scaling after 8 threads anyway (probably due to being bandwidth-limited). As can be seen by this Table, a parallel build scales extremely well to 2 threads, and continues scaling, but with worsening efficiency, up to 8 threads. In particular, we can build a fully SAH-based BVH faster than a single-threaded BIH-build. Still, when properly parallelizing the BIH build as well, for the same number of threads a BIH build is still about twice as fast as a SAH build, so the original trade-off between build time and BVH quality remains at least to a certain degree (the ratio in build times dropped from 20:1 to 2:1, but did not entirely disappear).

scene	#threads	Fairy Forest 2	BART	Expl. Dragon
grid+BIH	2	75ms (2.9×)	41ms (2.5×)	29ms (2.4×)
SAH+SAH	2	196ms (1.9×)	154ms (2.3×)	97ms (1.9×)
grid+BIH	4	61ms (3.5×)	27ms (3.9×)	19ms (3.7×)
SAH+SAH	4	120ms (3.0×)	95ms (3.7×)	58ms (3.2×)
grid+BIH	8	53ms (4.1×)	19ms (5.5×)	14ms (5.0×)
SAH+SAH	8	84ms (4.3×)	58ms (6.0×)	45ms (4.1×)

Table 2
Build times when building on 2, 4, and 8 threads. Grid+BIH is spatial median throughout, SAH+SAH uses a SAH for both coarse partitioning and for each subtree. Numbers in parenthesis denote speedups over single-threaded building as given in Table 1.

6 Asynchronous Dynamic BVHs

Even with a fast and parallel BVH builder, two problems remain. First, due to bandwidth limitations the parallel builder does not scale beyond a certain number of CPUs, leaving all other CPUs idle. Second, the absolute build time still places an upper limit on the frame rate that can be achieved.

Therefore, we propose to never wait for rebuilds, and instead perform all rebuilds *asynchronously* to normal rendering: while a new BVH is built on an application-specified number of K rebuild threads, the remaining $N-K$ threads proceed with rendering by deforming the most recently finished one. As soon as a new BVH is available, it replaces the currently used one. On a machine with N cores, this results in K dedicated rebuild threads, and $N-K$ dedicated update/render threads (see Figure 1). The actual number of rebuild threads currently has to be specified by the user upon startup; changing the number of rebuild threads dynamically during runtime (ideally in an automated way) provides for some interesting avenue of future work, but is not currently supported.

To allow multiple threads to work asynchronously on the same data, we have to double buffer the shared data, which consists of the vertex positions and, of course,

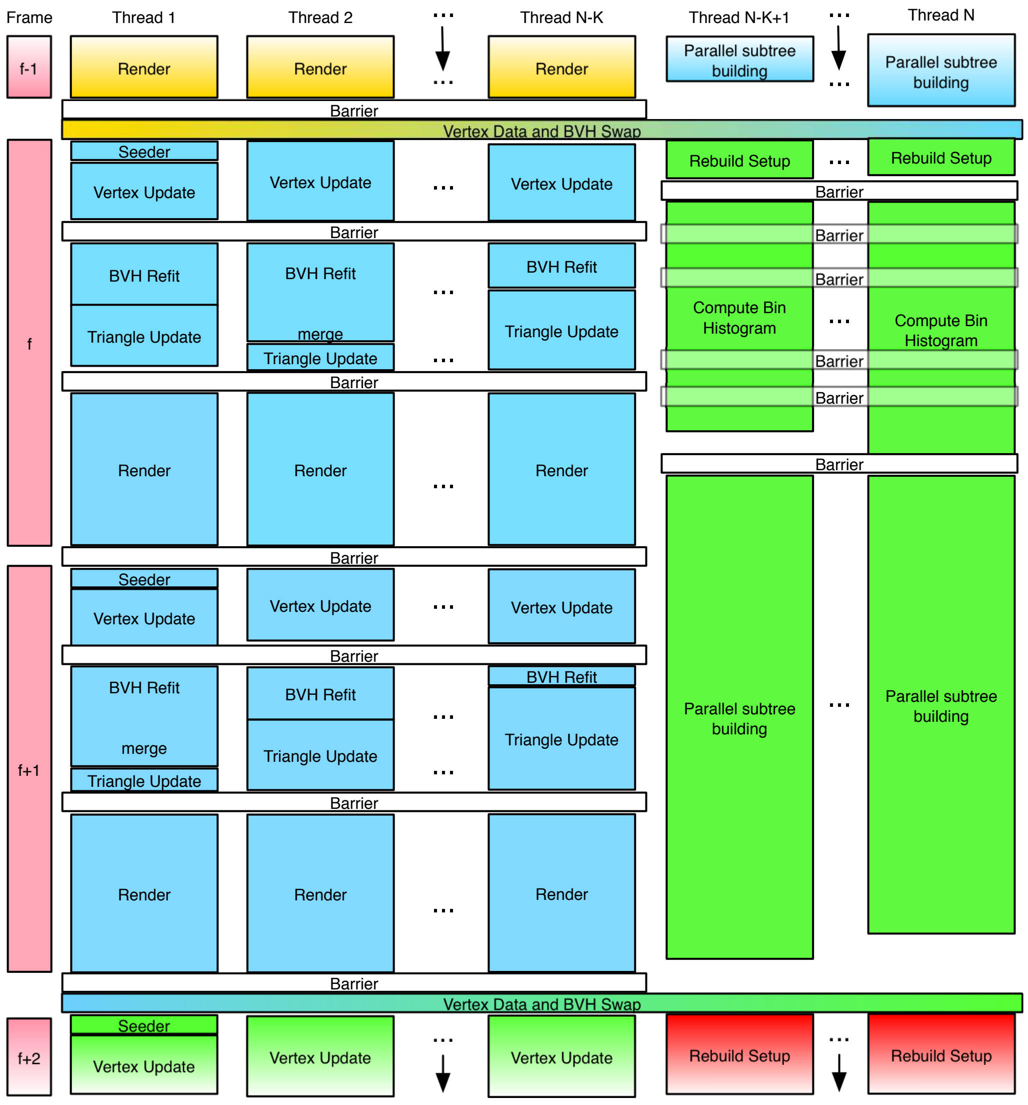

Fig. 1. Given a highly parallel architecture with N cores, $N-K$ of the cores work on parallel rendering and BVH refitting of the most recently finished BVH, while the K other cores work asynchronously and build in parallel the new BVHs as fast as possible, potentially over multiple frames (2 in this example). BVHs are deformed for only a few frames, and both scalability bottlenecks and pauses are avoided altogether.

the BVH nodes. All other data, like triangle connectivity, triangle acceleration structures, vertex normals, texture coordinates, etc are not touched by the builder, and so are not replicated. This results in roughly 80 bytes extra storage per triangle, which for most scenes has a minor impact. A large 1M triangle scene, for instance, would only require roughly 80MB of extra storage.

Whenever a new BVH is finished, the (parallel) rebuilder passes it to the $N-K$ rendering threads, and grabs a new set of vertices to work on. This naturally occurs between when the render threads finish their current frame and before they start refitting the BVH for the next frame. Since at that time the application has not yet computed the new vertex positions, we start the build process with vertex positions that are already one frame out-dated. While we could wait for the new vertex positions to be calculated before exchanging the data, this would require an expensive copy of those values to the rebuilder, which is especially problematic since the render threads must be blocked waiting for the copy to finish before they can use the new data. This critical section would hurt the system's scalability. Instead, by building the BVH from the last finished frame's vertex positions, the vertex and BVH buffers can be switched quickly with two pointer swaps, and the builder, application, and render threads can immediately continue. Furthermore, since it will likely take several frames before the new BVH is available anyway, that BVH will already be outdated by the time it is finished, so building the BVH with vertex positions that are outdated by one frame is equivalent to the build taking one frame longer to complete—which is a minor cost for ensuring good scalability.

With the BVH build decoupled from all other per-frame computations, the rendering stage itself can be kept highly parallel. In particular, the following operations are being performed by the render and update threads.

Vertex Generation. Vertices are usually generated by the application using, for example, a vertex shader or linear interpolation. Since even generating the vertices can be costly compared to refitting a BVH or rendering a frame, ensuring good system scalability requires parallelizing the vertex generation, too. In our current framework, we compute vertices by linearly interpolating between fixed timesteps, which we do in parallel on the $N-K$ update/render threads.

Parallel BVH Refit. Once the new vertex positions are known, we refit the most recently finished BVH's bounding volumes in parallel. To ensure scalability we use a three-way dynamic load-balancing scheme for the update. In the first phase, one "seeder" thread traverses the upper k levels of the BVH and records the node IDs of all the leaf nodes encountered and the node IDs of the k'th level subtrees. Though no other thread can start refitting until this seeding is done, there is no scalability issue as the seeding is extremely fast.

Once all the $N-K$ render/update threads are done updating the vertices and seeding, they synchronize on a barrier, and then switch to BVH refitting. We dynamically load-balance by having each thread take a node ID from the list, refit that subtree, and repeat. As soon as a thread finds no more subtrees to refit, it immediately goes on to performing triangle updates. The last thread to *finish* a subtree update also performs the final "merge" of the refit subtrees.

Triangle Update. For ray-triangle intersection, we use the method outlined in [26]. This method uses a precomputed set of data values for each triangle, which for an animated scene has to be recomputed every frame. Due to imbalances in the BVH refitting phase, the update threads can enter that phase at different times. We compensate by dynamically load balancing the triangle updates. In this way, all of the individual operations—parallel subtree update, serial subtree merge, and triangle update—are fully interleaved, ensuring that all rendering threads remain constantly utilized and finish at the same time.

Parallel Rendering. Once all update/render threads have finished updating, they synchronize themselves via a barrier, and then render the scene using a standard tile-based dynamic load balancing scheme. All per-frame operations—update and render—are dynamically load-balanced at all stages, and only three barrier operations are performed per frame: after vertex updates, after all threads have completed updating, and once all tiles have been rendered. The only non-parallelizable stage is the time between the end of the current and the start of the next frame, in which the application processes user input, displays the image, and if applicable, swaps the rebuild data. Even then, parallel rebuilding is active in the background.

BVH Build Method. The choice of BVH build method–as well as the number K of threads used for rebuilding–is completely orthogonal to the asynchronous rendering approach. When building asynchronously, a BVH will *always* be outdated by as many frames as it took to build this BVH. Thus, there is still a trade-off between a build method's resulting BVH quality and the time to achieve that quality, as longer builds potentially suffer worse from deterioration. Similarly, there is a trade-off related to how many threads are used for rebuilding, with more threads reducing the amount of BVH deterioration, but also takes resources away from rendering. These trade-offs we will investigate in more detail below.

7 Results and Discussion

Although mainly designed for upcoming multi-core architectures that will likely contain a large number of cores, current processor architectures only feature 2 to at most 4 cores per processor. We therefore use an 8 processor dual-core Opteron 8222SE shared-memory PC for our experiments, as that gives us a total of 16 cores, which we believe more closely resembles the number of cores on future hardware. Unless otherwise noted, we use all 16 cores in our tests. We use three test scenes: the “Fairy Forest 2”, the “Exploding Dragon”, and the “2×2 BART

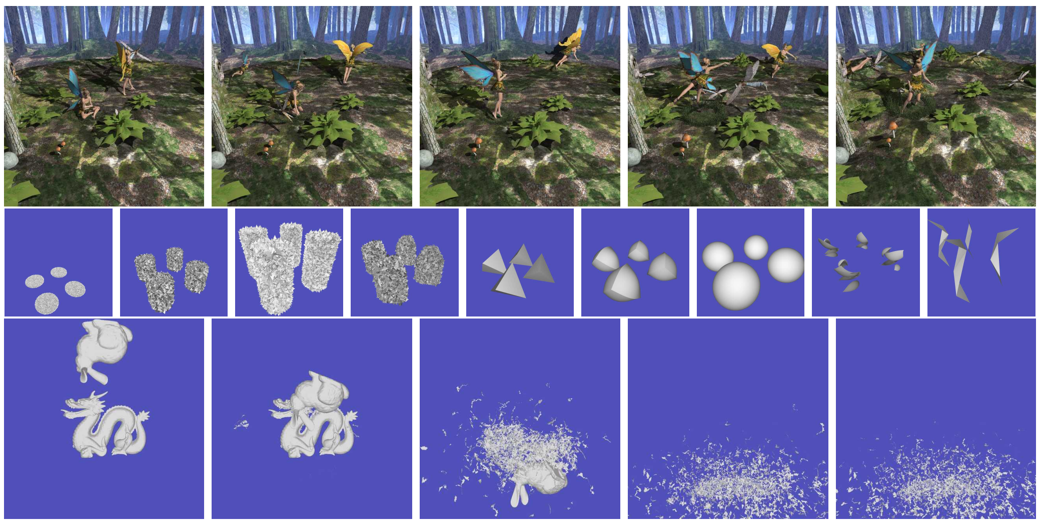

Fig. 2. The three scenes used for evaluating how our system performs for various kinds of motion. Top: “Fairy Forest 2” with 394K triangles. Middle: “BART 4” (part of the BART benchmark replicated 2x2 times), showing intentionally incoherent motion of 262K triangles. Bottom: The UNC “exploding dragon” of 252K triangles.

museum" (see Figure 2). The Fairy Forest 2 is a 7.75s long keyframed animation with 394K triangles, almost all of which are slowly deforming every frame, and resembles a game-like scene. The 252K triangle exploding dragon scene is 3.2s long and exhibits complex deformations occurring very quickly. The 262K triangle 2×2 BART museum is 8s long and composed of 4 copies of the museum scene from the Benchmark for Animated Ray Tracing (BART) [12] and is intentionally designed to stress test large deformations. Because the BART scene deforms heavily by morphing into wildly varying shapes (see Figure 2), it provides a challenge where standard BVH refitting quickly breaks down (see [11,22]). Finally, the UNC exploding dragon also shows incoherent motion, but in a less artificial setting.

All measurements were performed using the packet/frustum ray tracer used in [22]. To simulate the effect of a somewhat higher render-to-update cost ratio, the Fairy Forest 2 is rendered at 2048×2048 with lambertian shading and hard shadows. The other two scenes are rendered at 1024×1024 pixels with no shading.

7.1 Comparison to Synchronous Approaches

The first obvious question to investigate is how our asynchronous approach compares to traditional synchronous approaches. In Figure 3, we compare our method to the three standard approaches of (a) not rebuilding at all ("refit only", as used in, e.g., [22]), (b) rebuilding every frame as fast as possible ("full rebuild", [20]), and (c) rebuilding infrequently as indicated by the "rebuild heuristic" as proposed in [11]. For both the full rebuild and the rebuild heuristic, we use the fast, parallel BVH build described before. We found that using 8 instead of all 16 cores gave the fastest rebuild times, and so use 8 threads for these (synchronous) rebuilds.

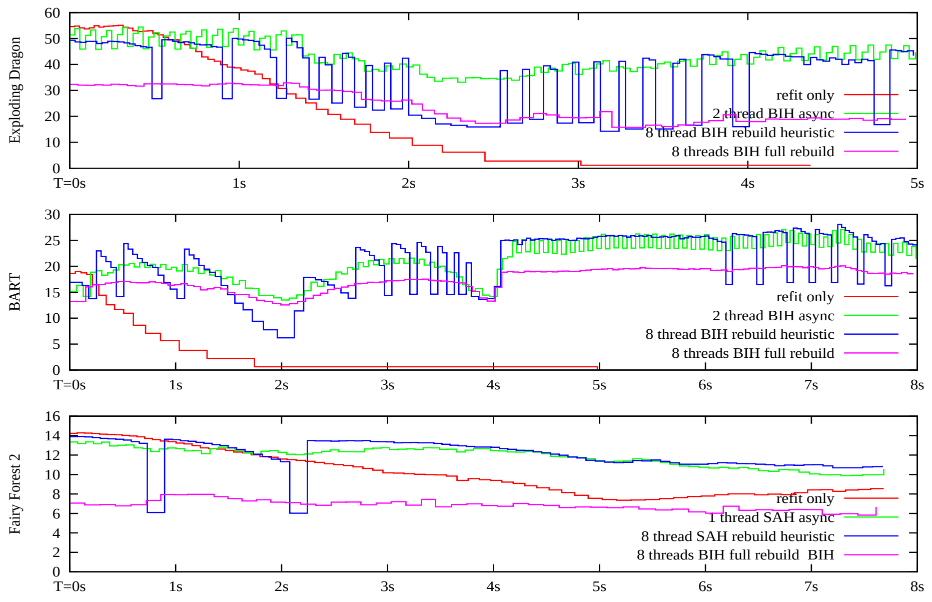

Fig. 3. Impact on frame rate using various build strategies. For the BART and Exploding Dragon scenes, 2 thread BIH performed best for the asynchronous build, while 1 thread SAH performed best for the Fairy Forest 2 scene.

As can be seen from Figure 3, all scenes show that simply refitting leads to severe performance deterioration, with about a 1.5× drop for the Fairy scene, and a nearly complete standstill for the BART and dragon scenes. Lauterbach's approach is clearly superior to refitting only; it avoids the BART scene's extreme deterioration, and achieves higher frame rates for the Fairy scene. Furthermore, with only two rebuilds triggered for the Fairy scene, it achieves nearly optimal frame rates for most of the animation.

While these experiments confirm Lauterbach's rebuild heuristic is superior over deforming only, they also show its weaknesses: the high variation in frame rate caused by rebuilds and varying rates of deterioration, the "sawtooth" effect of deterioration until a new build is triggered, and, in particular, the disruptions in which the system temporarily freezes when a new BVH is being built. Per-frame rebuilding avoids this effect, and produces the smoothest frame rate. However, it usually exhibits the lowest frame rate of all the methods, even when parallelized.

Compared to these methods, our method is clearly advantageous: Though the rebuild heuristic or refitting only can reach higher peak performance (because less cores are used for rendering), our method is clearly competitive where these techniques are best, and clearly outperforms per-frame rebuilding. In addition, our technique suffers from neither accumulated deterioration (as in refitting) nor from severe sawtooth effects and disruptions (as with the rebuild heuristic).

7.2 *Fast vs. Slow BVH Building*

As mentioned above, asynchronous rebuilding can decouple build time from render time, but the longer the build takes, the more BVH deterioration can accumulate in the currently refitted BVH. Obviously this effect depends on how severely the scene deforms (i.e., on the speed of the animation and the kind of motion). To quantify this effect, we have measured the frame rate for both the original sweep SAH build, the fast binned build, and a BIH build (each using one thread).

The results of this comparison are given in Figure 4: Though the sweep build should produce the best BVH quality, it consistently achieves much lower frame rate than the BIH build or the fast binned build, as too much deterioration has accumulated by the time the BVH is built. Note that this is true even for the first frame rendered with a newly available BVH (around $t = 6s$), as that newly available BVH is already

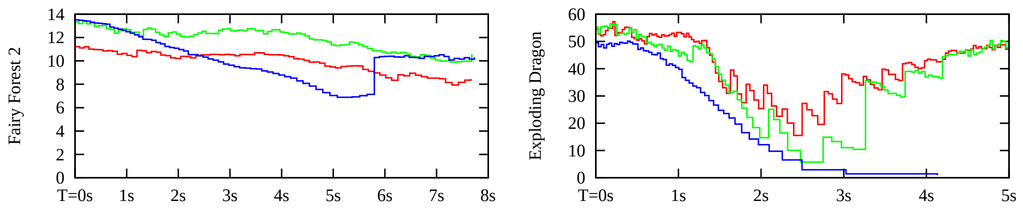

Fig. 4. Impact of build time in the asynchronous system for single thread BIH, binning, and sweep SAH. For the fairy with high render-to-rebuild cost and smooth deformation, the SAH build wins, while for the dragon scene with very simple shading, the BIH build performs at least as good. Both BIH and fast SAH are clearly superior to the sweep build.

outdated. If deformation is as severe as in the dragon or BART scenes, slow building results in a severe sawtooth effect, or even a complete breakdown in performance.

7.3 Relation on Build time and Frame Rate

It is important to realize, however, that as long as the animation is specified in world time (as is usually the case) this sawtooth effect is *not* related to render performance at all, but only to how much world time has to be bridged with deformation. Higher or lower render performance via different hardware or shaders, for example, would change the number of frames rendered during the time needed to build the new BVH, but the amount of deterioration accumulated in that time (and thus, the relative performance impact) would not change.

7.4 Number of Threads and Choice of Build Method

Section 7.2, showed that reducing the build time can be crucial for avoiding the accumulation of deterioration. On the other hand, Section 7.3 demonstrated that it is only the *absolute* build time that matters, not its relation to render time—which suggests that once a certain rebuild performance has been reached, either through more cores, faster cores, or better build algorithms, building even faster will have a diminishing return.

Thus, the exact balance on how many threads and which build method to use cannot be answered without knowing what the actual hardware and scene characteristics are. Generally, for scenes with slow smooth deformations, a single build thread may suffice, while for scenes with many triangles or with severe deformations, using more threads for rebuilding is advantageous even if that takes resources away from rendering.

Similarly, the question on whether to use the BIH build or an SAH build remains hard to answer. Even the fast SAH build takes around twice as long as a BIH build, so for the same absolute build time using a BIH build would free more threads for rendering. For the same number of build threads, the BIH build would finish sooner, and the reduced amount of accumulated deterioration can easily offset the lower quality trees in rapidly deforming scenes. Furthermore, the BIH build does not subdivide nodes as much as an SAH build, and while this does lead to fatter leaves and slower rendering, it also makes both building and refitting faster. For simple scenes, the refitting time can be equivalent to the rendering time (the exploding dragon scene takes 11ms to refit and 6-13ms for rendering), so the improved refitting time can easily compensate for the slower rendering time. Note that the parameters of an SAH build can also be modified to produce shallower trees.

Thus, it is impossible to come up with a configuration that is the best for all scenes and hardware configurations. For our current 16-core system, we usually use one or two threads for rebuilding, and would likely use up to 4 build threads if the scenes would get larger and the deformations more severe. As to build method, we usually use the fast SAH for low deforming scenes (usually the common case), and the BIH build when the rate of deformation is high. With this configuration, our system was well able to handle all the animated models we have so far tested, and usually achieves a system utilization of at least 90%.

7.5 Remaining limitations

While we significantly reduce the dependence on refitting, we still refit for at least one frame. Thus, completely unstructured motion with near-randomly changing geometry every frame cannot be supported. However, practical applications for such completely random scenes are probably rare, and more likely effects, such as exploding objects, are arguably not worse than what happens in the BART and exploding dragon scenes, which our method handle well.

Similarly, relying on refitting does not allow for changing scene topology. However, this usually occurs only if entire objects appear into/disappear from a composite scene environment, which can easily be handled by a two-level approach as proposed in [21]: the small top-level scene could easily be rebuilt per frame, with our method handling the per-object deformations. In cases where completely new geometry is required—e.g., when a player enters a new level, or passes a portal—one could perform a single, parallel but synchronous rebuild. However, if known in advance, the rebuild could also be done asynchronously so that by the time it is required it will already be built. In fact, if rebuilds are fast enough to occur per frame, and we have found that they can be with 2-4 rebuild threads, then we could even handle completely dynamic scenes by rendering the BVH built during the previous frame, which is analogous to the double buffering used in rasterization. This is still advantageous since the overall frame rate is not the sum of the rebuild and render time, but just the max of the two.

8 Conclusion

In this paper, we have presented a new approach to handling dynamic scenes in a highly parallel ray tracing system. Instead of trying to do a full BVH rebuild per frame, we rebuild asynchronously over the course of one or more frames, and in the meantime rely on refitting, which parallelizes well.

Our method's advantage over preexisting methods depends on the amount of deformation in a scene. If the deformation is sufficiently small, simply refitting every frame may suffice, rendering our method superfluous.

Since we still depend on a scene's deformability for at least short periods of time, we cannot handle randomly deforming scenes, or scenes with changing topology as ably as other data structures with faster complete rebuilds, such as a grid [24]. However, most scenes do not require full rebuilds each frame, and for those that do, we can still get up to twice the performance of previous BVH ray tracers by completely rebuilding the next frame's BVH while rendering the current frame.

Our methods are particularly designed for highly parallel multi-core architectures, be it CPU cores, GPU cores, CELL SPEs, or even special purpose hardware. While increasing parallelism is a problem for pure rebuilding, our methods in fact benefit from more cores, as the relative overhead decreases. As argued in the previous section, the currently foreseeable trends towards having many more cores, slightly more performance per core, and more rays per pixel would make our method even more suitable for these architectures than the one we used in our experiments.

Arguably the biggest limitation is that we have no way of predicting which configuration will work best for any given scene. Determining heuristics for doing that—potentially on the fly—would be an interesting avenue for future work.

References

[1] J. Amanatides and A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics '87*, pages 3–10. Eurographics Association, 1987.

[2] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Technical report, University of Texas at Austin Dep. of Comp. Science, 2007. Conditionally accepted to ACM Transactions on Graphics.

[3] S. A. Green and D. J. Paddon. A Highly Flexible Multiprocessor Solution for Ray Tracing. *The Visual Computer*, 6(2):62–73, 1990.

[4] M. Harris and D. Luebke, editors. *Supercomputing Tutorial on GPGPU*, 2006.

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

[6] W. Hunt, W. R. Mark, and D. S. Fussell. Fast, Lazy Acceleration Build From Hierarchy. In *Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing*, 2007.

[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*, 2006.

[8] T. Ize, I. Wald, and S. G. Parker. 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.

[9] T. Ize, I. Wald, C. Robertson, and S. G. Parker. An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 47–55, 2006.

[10] F. Jansen. Data structures for ray tracing,. In *Proceedings of the Workshop in Data structures for Raster Graphics*, pages 57–73, 1986.

[11] 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*, pages 39–45, 2006.

[12] J. Lext, U. Assarsson, and T. Möller. BART: A Benchmark for Animated Ray Tracing. Technical report, Department of Computer Engineering, Chalmers University of Technology, 2000.

[13] M. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium*, 1995.

[14] S. G. Parker, W. Martin, P.-P. Sloan, P. Shirley, B. E. Smits, and C. D. Hansen. Interactive ray tracing. In *Proceedings of Interactive 3D Graphics*, 1999.

[15] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[16] A. Reshetov, A. Soupikov, and J. Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics (Proceedings of ACM SIGGRAPH)*, 24(3):1176–1185, 2005.

[17] S. Rubin and T. Whitted. A 3D representation for fast rendering of complex scenes. In *Proceedings of SIGGRAPH*, pages 110–116, 1980.

[18] M. Shevtsov, A. Soupikov, and A. Kapustin. Fast and scalable kd-tree construction for interactively ray tracing dynamic scenes. *Computer Graphics Forum (Proceedings of EUROGRAPHICS)*, 26(3), 2007.

[19] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006 – Proceedings of the 17th Eurographics Symposium on Rendering*, pages 139–149, 2006.

[20] I. Wald. Fast and Parallel Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing*, 2007.

[21] 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*, pages 11–20, 2003.

[22] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):1–18, 2007.

[23] 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*, pages 61–70, 2006.

[24] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 25(3):485–493, 2006.

[25] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In *State of the Art Reports, Eurographics 2007*, 2007.

[26] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS)*, 20(3):153–164, 2001.

[27] S.-E. Yoon, S. Curtis, and D. Manocha. Ray Tracing Dynamic Scenes using Selective Restructuring. In *Eurographics Symposium on Rendering*, 2007.