RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs

Christian Lauterbach*
University of North Carolina at Chapel Hill

Sung-Eui Yoon†
Lawrence Livermore National Laboratory

David Tuft‡
University of North Carolina at Chapel Hill

Dinesh Manocha§
University of North Carolina at Chapel Hill

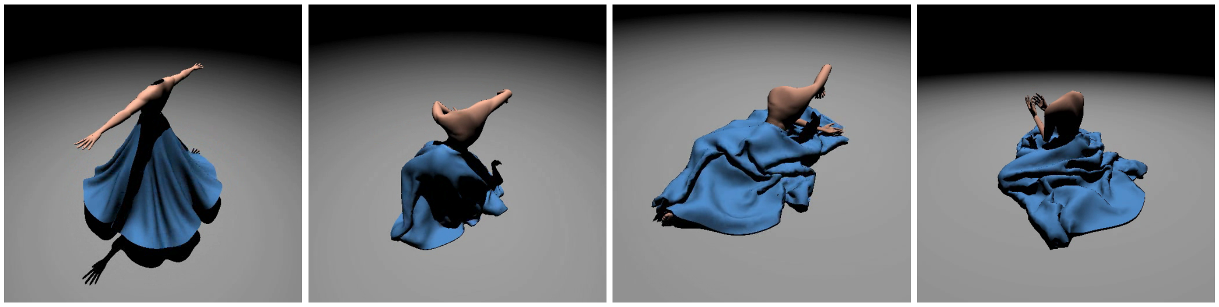

Figure 1: **Dress simulation**: *Four different images of a* 210 *step sequence taken from a dynamic cloth simulation and consisting of* 40*K triangles. By updating in real-time instead of rebuilding the BVH of the deforming model according to our heuristic, we are able to render the animation at* 13 *frames per second with* 512^2 *screen resolution using a dual-core P4 processor at 2.8 GHz.*

Abstract

We present an efficient approach for interactive ray tracing of deformable or animated models. Unlike many of the recent approaches for ray tracing static scenes, we use bounding volume hierarchies (BVHs) instead of kd-trees as the underlying acceleration structure. Our algorithm makes no assumptions about the simulation or the motion of objects in the scene and dynamically updates or recomputes the BVHs. We also describe a method to detect BVH quality degradation during the simulation in order to determine when the hierarchy needs to be rebuilt. Furthermore, we show that the ray coherence techniques introduced for kd-trees can be naturally extended to BVHs and yield similar improvements. Finally, we compare BVHs to spatial kd-trees, which have been used recently as a replacement for AABB hierarchies. Our algorithm has been applied to different scenarios arising in animation and simulation and consisting of tens of thousands to a million triangles. In practice, our system can ray trace these models at 3-13 frames a second on a desktop PC including secondary rays.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: ray tracing, bounding volume hierarchies, deformable models, animation

1 Introduction

Ray tracing is a classic problem in computer graphics and has been studied in the literature for more than three decades. Most of the earlier ray tracing algorithms were used to generate high quality images for offline rendering, but int the last few years, there has been renewed interest in real-time ray tracing. At a broad level, most of the work in real-time ray tracing algorithms can be classified into three main categories: improved techniques to compute acceleration structures, exploiting ray coherence, and parallel algorithms on shared memory or distributed memory systems.

Most current interactive ray tracing algorithms use kd-trees as an acceleration data structure [22,31]. In practice, kd-trees are simple to implement, can be stored in a compact manner, and are used for efficient tree traversal during ray intersections. However, one of the the main disadvantages of kd-trees is the high construction time; current algorithms can take seconds even on models composed of tens of thousands of triangles [9,30]. Although fast build methods exist (e.g. [37]), the relative performance penalty for using a faster kd-tree build algorithm is severe compared to doing the same with object hierarchies. Furthermore, no simple and fast algorithms are known for incrementally updating the kd-tree hierarchy, even when the primitives undergo a simple deformation. As a result, current algorithms for interactive ray tracing are mainly limited to static scenes.

Main results: In this paper, we present a simple and efficient algorithm for interactive ray tracing of dynamic scenes of deformable objects, i.e. where the number of primitives does not change. We analyze many issues with respect to computation and incremental updates of hierarchies. Our algorithm uses bounding volume hierarchies (BVHs) of axis-aligned bounding boxes (AABBs), for which we describe efficient techniques to recompute or update these hierarchies during each frame. In practice, rebuilding of BVHs can be expensive, so we minimize these computations by measuring BVH quality degradation between successive frames. We also apply the ray coherence techniques developed for kd-trees to BVHs and obtain similar speedups. Finally, we describe techniques to parallelize these computations on multi-core architectures and improve the cache efficiency of the resulting algorithms. We have implemented our algorithm and highlight its performance on several dynamic scenes. Our system can render these datasets with

*e-mail: cl@cs.unc.edu
†e-mail:sungeui@llnl.gov
‡e-mail:tuft@cs.unc.edu
§e-mail:dm@cs.unc.edu

secondary rays at $3-13$ frames per second on a dual-core desktop PC.

Overall, our approach offers the following advantages:

1. **Simplicity:** Our algorithm is very simple and easy to implement.
2. **Interactivity:** We are able to handle dynamic scenes with up to a million triangles at interactive rates on current desktop PCs.
3. **Generality:** Our algorithms make no assumptions about the motion of the objects or the underlying simulation or animation, as long as no primitives are added or deleted.

The rest of the paper is organized in the following manner: We give a brief overview of previous methods in Section 2. We present our BVH hierarchy computation algorithm and evaluate its features with other approaches in Section 3. Section 4 describes our ray tracing algorithm for dynamic scenes based on BVHs and addresses the issue of utilizing multi-core architectures. We show the results obtained by our approach on several benchmarks in section 5. Finally, we compare our method to a recent related approach in section 6.

2 Previous Work

In this section, we give a brief overview of prior work in interactive ray tracing and dynamic scenes.

Interactive ray tracing: Since its early introduction in [1,38], the ray tracing algorithm has been very well studied in computer graphics due to its generality and high rendering quality. Several systems have been presented that are capable of generating ray traced images at interactive speeds. A recent survey is given in [25]. Parker *et al.* [20] present a real-time ray tracing algorithm on a shared-memory supercomputer. Several approaches use ray coherence to improve performance and achieve interactive performance on commodity desktop systems for large static datasets, such as coherent rat tracing [31,33]. Recently, MLRT [22] combines kd-tree traversal with frustum culling to further improve performance. Additionally, level-of-detail approaches have been used to improve the performance of ray tracing massive models [3,12,41].

Dynamic Scenes: There is relatively less work on ray tracing dynamic scenes. Reinhard et al. [21] use a grid structure that can be updated efficiently for any type of animation. Lext *et al.* [16] present a general purpose framework and benchmarks for ray tracing animated scenes. They also propose an algorithm that uses oriented bounding boxes along with regular grids [17]. Wald et al. [32] describe a distributed system for dynamic scenes that differentiates between transformations and unstructured movement in the scene. Recently, Wald *et al.* [36] proposed a coherent grid traversal algorithm to handle dynamic models. Another current approach keeps the kd-tree, but uses the bounds of the primitives over the whole animation [8] so that the kd-tree structure is correct for each frame, at the cost of increased traversal overhead.

Bounding volume hierarchies: BVHs have been widely used to accelerate the performance of ray tracing algorithms [23, 26]. In the case of static scenes, algorithms based on kd-trees and nested grids seem to outperform BVH-based algorithms [9]. Larsson and Akenine-Möller [15] present a lazy evaluation and hybrid update method to efficiently update BVHs in collision detection. They also use the algorithm to ray trace models composed of tens of thousands of polygons [14]. BVHs have also been used to accelerate the performance of collision detection algorithms for deformable models [28, 29]. Interactive ray tracing using BVHs has been demonstrated by Geimer and Müller [5] as well as Wald *et al.* [34]. The latter also use BVH updates to handle animated scenes. The main difference to our system is that we do not assume advance knowledge of the animation in order to find the best hierarchy, but are able to detect when the hierarchy needs to be recomputed. Finally, Boulos *et al.* [2] demonstrate interactive distribution ray tracing on a medium-sized shared memory system.

3 BVHs for dynamic scenes

In this section, we analyze the problem of ray tracing using BVHs. We show that BVHs can offer better performance than kd-trees on dynamic environments and present optimizations to speed up rendering.

3.1 Choice of Hierarchies

A BVH is a tree of bounding volumes. Each inner node of the tree corresponds to a bounding volume (BV) containing its children and each leaf node consists of one or more primitives. Common choices for BVs include spheres, AABBs, oriented bounding boxes (OBBs) or k-DOPs (discretely oriented polytopes). Many efficient algorithms have been proposed to compute sphere-trees [11], OBB-trees [7], and k-DOP-trees [13]. However, we use AABBs as the BV as they provide a good balance between the tightness of fit and computation cost and employ efficient algorithms for ray-box intersection [19, 24, 39].

3.2 AABB hierarchies vs. kd-trees

In this section, we evaluate some features of BVHs based on AABBs and compare them with kd-trees for ray tracing. Recently, many efficient and optimized ray tracing systems have been proposed based on kd-trees [31]. As far as static scenes are concerned, analysis has shown that optimized algorithms based on kd-trees will outperform BVH-based algorithms [9]. There are multiple reasons to explain this behavior: First, even the most optimized ray-AABB intersection test (e.g. from [39]) is more expensive than split plane intersection for kd-trees. This is due to the fact that in the worst case (i.e. no early rejection) up to 6 ray-plane intersections need to be computed for AABB trees, as opposed to just one for a kd-tree node. Another important aspect is that a BVH does not provide real front-to-back ordering during traversal. As a result, when a primitive intersects the ray, the algorithm cannot terminate (as is the case for a kd-tree), but needs to continue the traversal to find all other intersections. Furthermore, kd-tree nodes can be stored more efficiently (8 bytes per node [35]) than an AABB possibly could. On the other hand, we found that BVHs often need fewer nodes overall to represent the scene as compared to a kd-tree (please see Table 1). This is mainly due to the fact that primitives are referenced only once in the hierarchy, whereas kd-trees usually have multiple references because no better split plane could be found. In addition, AABBs have the advantage of providing a tighter fit to the geometric primitives with fewer levels in the tree, e.g. kd-trees need multiple subdivisions in order to discard empty space. Most importantly, the major benefit of BVHs is that the trees can be easily updated in linear time using incremental techniques. No similar algorithms are known for updating kd-trees.

3.3 BVH Construction

We construct an AABB hierarchy in a top-down manner by recursively dividing an input set of primitive into two subsets until each subset has the predetermined number of primitives. We have found that subdividing until each leaf just contains one primitive yields the best results at the cost of a deeper hierarchy, as – similar to kd-trees – node intersection is comparably cheaper than primitive intersection, although other authors have reported best performance for 6 primitives per node [18]. During hierarchy construction, the most important operation is to find a divider for the two subsets that will

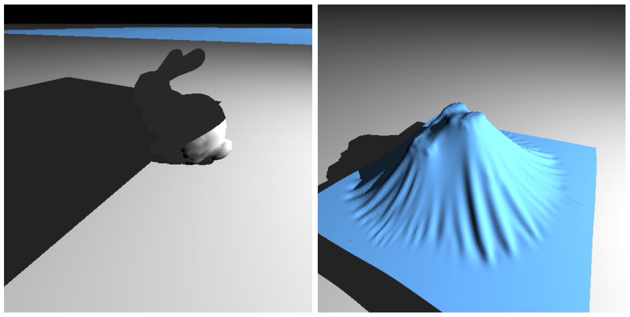

Figure 2: **Cloth on Bunny Simulation:** *Two shots of a* 315 *step dynamic simulation of cloth dropping on the Stanford bunny. We achieve* 13 *frames per second at* 512^2 *screen resolution using a dual-core P4 processor at 2.8 GHz.*

optimize the performance of runtime ray hierarchy traversal. One of the best known heuristics for tree construction for ray tracing is the *surface-area heuristic* (SAH) [6, 9], which has been shown to yield higher ray tracing performance. However, despite recent improvements [30], it also has a much higher construction cost and will commonly take longer than the actual frame rendering time for dynamic environments. Because of this, we use the spatial median of one of the dimensions and sort the primitives into the child nodes depending on their location with respect to the midpoint. We observe that the spatial median build is usually about an order of magnitude faster to compute and provides rendering performance of 50-90% of SAH for most scenes. Note that even though we just split along one dimension, the bounding box will still be tight along all the three dimensions. As this method will often distribute a similar number of primitives to both children, the resulting tree will likely be nearly balanced. As we are storing just one primitive per leaf, it is also easy to see that the total number of nodes in the tree for n primitives will always be $2n-1$, which allows us to allocate the space needed for any subtree during construction.

Regardless of the heuristic for finding a split, the time complexity, $T(n)$, of the top-down AABB hierarchy construction algorithm is $\Omega(n \log_k n)$, where k is the number of children of each node and n the number of primitives. It is easy to see that for each split, every primitive in the node needs to be processed at least once to see which child it belongs in. Since at each level of the tree during construction all n primitives are examined and the smallest possible number of levels is $\log_k n$, any top-down construction has to take at least $\Omega(n \log_k n)$ time.

3.4 Updating the hierarchy

The main advantage of using BVHs for ray tracing is that animated or deforming primitives can be handled by updating the BVs associated with each node in the tree. Our algorithm makes no assumptions about the underlying motion or simulation. In order to efficiently update the hierarchy, we recursively update the BVHs by using a postorder traversal. We initially traverse down to leaves from the root nodes. As we encounter a leaf node, we efficiently compute a new BV that has the tightest fit to the underlying deformed geometry. As we traverse from the leaf node in a bottom-up manner, we initialize the BV of an intermediate node with a BV of the leftmost node and expand it with the BVs of the rest of the sibling nodes.

The time complexity of this approach is $O(n)$, which is lower than the construction method. This is reflected by update times that we have found to be about 4 times faster than rebuilding the tree for our benchmark models (see Table 1 for detailed results). Therefore, we rely on hierarchy update operations to maintain interactive performance for dynamic environments.

3.5 BVHs for deformable scenes

We initially build an AABB tree of a given scene. As the model deforms or some objects in the scene undergo motion, the BVH needs to be updated or rebuilt. Updating the BVH is to recompute the bounds of each BV node, and rebuilding the BVH is to recompute the entire BVH from scratch and re-clustering the primitives. At runtime, we traverse the BVH to compute the intersections between the rays and the primitives.

If the algorithm only updates the BVH between successive frames, the runtime performance of BVHs can degrade over the animation sequence because the grouping of the primitives and structure of the hierarchy does not change. As a result, the BVs may not provide a tight fit to the underlying geometric primitives. This is often characterized by growing and increasingly overlapping BVs, which subsequently deteriorate the quality of the BVH for fast runtime BVH traversal by adding more intersections between the ray and AABBs. In such cases, rebuilding the AABB tree or parts of it is desirable.

We found that updating the BVH works well with relatively small changes to the scene or structured movement to groups of primitives such as meshes. When primitives move independently, however, for example in different directions, changes to the actual tree structure may be necessary to reflect the new positions of the deforming geometry. Still, rebuilding the BVH can be considerably more expensive than updating the BVH. As a result, we want to minimize the number of times rebuilding is performed. Therefore, we need to efficiently decide when updating the BVH is sufficient or rebuilding the BVH is required. This is non-trivial because the actual degradation of a BVH depends on many factors, such as the speed with which primitives move and the general characteristics of the motion of objects in the scene. Simple approaches such as rebuilding the tree every t frames have the disadvantage of not being adaptable to different characteristics over the animation and need to be chosen a priori. Conservatively choosing t means adding a lot of rebuilding overhead, which is especially unwanted in an interactive context. In order to efficiently detect when updating tree or rebuilding tree is required, we use a simple heuristic that is described in the next section.

3.6 Rebuilding criterion

We assume that BVH quality degradation is marked by bounding box growth that is not caused by actual primitive size, but by distribution of primitives or subtrees in the box. For example, consider two primitives moving in opposite directions. The parent node containing them will have to grow to accommodate for the movement, resulting in a bounding box that is relatively large, but mostly empty. Since the probability that a box will be intersected by a ray rises with its surface area, we want to rebuild a subtree to find a more advantageous tree topology. To find these cases and prevent them from impacting performance, we need to measure BVH degradation during each frame by using a simple and inexpensive heuristic.

Our heuristic is based on the idea that we can find nodes that are large relative to their children by comparing their surface area. In order to have a relative metric independent of scale, we measure the ratio of each parent node's surface area to the sum of the area of its two children. The larger the ratio becomes, the more imbalance exists in the sizes. We first compute the ratio during tree construction and store it in a field of the optimized AABB data structure (see next section). Whenever the tree is updated, the changed surface areas are automatically computed and each inner node can easily calculate its new ratio. Since we assume that the ratio stored from the construction is as good as we can do, we find the difference be-

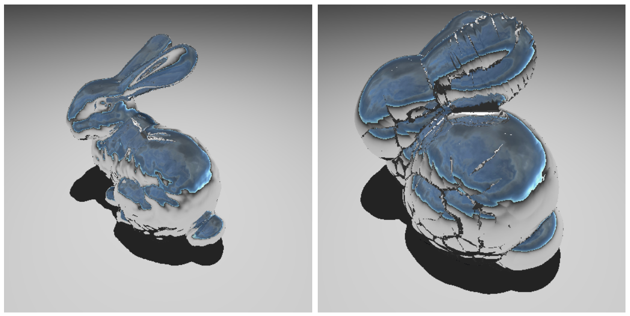
Figure 3: **Bunny blowing up :** *Two images show frames from a* 113 *step animation of a deforming Stanford bunny. We achieve an average* 6 *frames per second during ray tracing this deforming model with shadow and reflection rays at* 512^2 *screen resolution using a dual-core P4 processor at 2.8 GHz.*

tween the new and old ratio and add them to a global accumulation value. Once the bottom-up update reaches the root, we have computed the sum of all the differences. To assure that this value can be tested independently of the tree size, we normalize it by dividing by the number of nodes that contribute to the sum, i.e. the sum of inner nodes, which is always $n-1$. This yields a relative value describing the overhead incurred by updating the BVH instead of rebuilding it. This value is then simply compared to a predefined threshold value (we found a threshold of 40% to work well in our tests) and the tree is rebuilt if the threshold is exceeded.

This approach has several advantages: it will detect a good time to rebuild regardless of the actual frame rate and without any scene-specific settings. Furthermore, in scenes where there is little to no degradation, the heuristic will never need to initiate a rebuild. It is also possible to use the method to just rebuild subtrees, but we found that this cannot fully replace a complete rebuild since degradations in the upper levels of the hierarchy typically have the highest impact on the performance of ray tracing. Therefore, an implementation that rebuilds only subtrees will have to either do a full rebuild sometimes, or support a top-level update where only the upper levels of the tree are rebuilt.

4 Ray Tracing with BVHs

In this section we describe our runtime BVH traversal algorithm. Also, we present techniques to extend the algorithm to multi-core architectures.

4.1 Traversal and Intersection with BVHs

We use a simple algorithm to compute the intersection of a ray and the scene primitives using the BVH. The ray is checked for intersections with the children of the current node starting at the root of the tree. If it intersects the child BV, the algorithm is applied recursively to that child, otherwise that child is discarded. Whenever a leaf node is reached, the ray is intersected with the primitives contained in that node. For most rays, the goal is to find the first hit point on the ray, so even if a ray-primitive intersection is found, the algorithm has to search the other sub-trees for potential intersections. An exception to this are shadow rays, where (at least for directional lights) any hit is considered sufficient and traversal can stop.

BVH traversal optimizations: Experience with kd-trees has shown that front-to-back ordering is a major advantage for ray tracing. Although BVHs do not provide a strict ordering, we found that storing the axis of maximum distance between children for each AABB and using that information during traversal together with the ray direction to determine a 'near' and 'far' child improves the traversal speed, especially for scenes with a high depth complexity (this has also been reported in [18]. Another issue is cache coherence during traversal: similar to the compact kd-tree representations [35], we can optimize the AABB representation to fit within 32 bytes. We achieve this by storing the bounding box as 6 floating point values, one child pointer (such that the second child is expected to be directly after the first one in memory) and one float for storing quality information for the rebuild heuristic. Our profiling shows that BVH traversal using our AABBs has the same cache efficiency as the kd-tree traversal. To use an even more compact representations, other implementations also have used nodes where the actual coordinates were quantized and compressed to save memory (such as [4, 18]).

Use of ray coherence techniques: One of the main techniques used in current real-time ray tracers is to exploit ray coherence to reduce the number of traversal steps and primitive intersections per ray. Those algorithms were originally designed for the kd-tree acceleration structure. It is relatively straightforward to extend them to work with BVHs as well. In order to use coherent ray tracing [33] the BVH traversal has to be changed so that a node is traversed if *any* of the rays in the packet hits it and skipped if *all* of the rays miss it. A hit mask is maintained throughout the traversal to keep track of which rays have already hit an object and their distance. However, the traversal does no longer require that the rays have the same direction signs because unlike kd-trees the traversal order does not determine the correctness for a BVH. We have implemented ray packet traversal for 2×2 ray bundles using 4-wide SIMD instructions and found that it yields an overall speedup of about 2 to 3, which is even above the improvement obtained for kd-trees. We assume this is mainly because ray-AABB intersections are more costly than the kd-tree's ray-plane computation and therefore the reduction in traversal steps has a more pronounced effect on overall performance. Furthermore, we also support arbitrary-sized ray packets, which can be implemented very efficiently by using frustum culling such as presented in [22]. Depending on the detail level of the scene and the screen resolution, 16x16 or 8x8 packets will yield an even higher speedup to rendering and performs much better than the normal packet traversal code that tests each ray [18].

4.2 Multi-Core Architectures

One of major features of current computing trends is that there are multiple cores and hyper-threading functionality available on commodity architectures. Therefore, it is desirable to design our hierarchy update and runtime traversal such that they take advantage of available parallelism.

Hierarchy Update: Our update method takes advantage of multi-core processors by using a bottom-up update method. Given the number of available threads, n, we decompose an input BVH into n sub-BVHs. For this, we simply compute n different children by traversing the tree from the root in the breadth-first manner. Then, each thread performs a bottom-up update from one of the computed nodes in parallel. After all the threads are done, we then sequentially update the upper portion of the n nodes. We particularly choose the bottom-up approach since it is well suited to parallel processing. For example, we do not require any expensive synchronization for each thread since data that are accessed by threads are mutually exclusive to each other. Since our current BVHs are relatively well balanced, this simple scheme provides reasonably good load balancing in practice.

Runtime traversal: We employ image-space partitioning to allocate coherent regions to each thread. Also, in order to achieve reasonably good load balancing, we first decompose image-space into small tiles (e.g., 16×16) and, then, allocate each tile to each thread. After a thread finishes its computation, it continues to process another tile. A more elaborate tile distribution may be necessary when using highly-parallel machines [27], but we have found that this approach works well for workstation-class machines and provides perfect scaling.

5 Implementation and Results

In this section, we describe our implementation and highlight the results of our ray tracer on different benchmarks.

5.1 Implementation

We have implemented our interactive ray tracer for deformable models using BVHs in a dual-core Intel Pentium 4 machine at 2.8 GHz. To compare the performance of BVHs with previous interactive ray tracing work for rendering static scenes, we also implemented kd-tree rendering (without animation capability). All acceleration structures support ray packet traversal using the SSE SIMD instruction set on Intel processors. For efficiency reasons, we only support triangles as primitives. We employ multi-threaded rendering and hierarchy updates using OpenMP.

5.2 Results

We have tested our system on four animated scenes of varying complexity as well as one more complex static model to measure performance of our approach (see Tables 1 and 2). In general, building a BVH tree using the naive midpoint method is more than one order of magnitude faster than the optimized surface-area heuristic kd-tree construction. In most cases, both structures have a similar memory footprint, but kd-trees need more nodes because references to primitives can be located in multiple nodes.

Benchmarks: We show five different test cases (refer to Table 1): Bunny/Cloth (shown in Fig. 2) and Dress (shown in Figure 4 on colour plate) in the respective rows of the table demonstrate performance on a typical animation including simulated cloth at different complexity, both rendered including shadow rays. Even though most of the mesh is moving, BVH updates turn out to be sufficient to maintain the quality of the structure. Bunny (shown in Fig. 3) applies a non-rigid deformation to the Stanford bunny model with reflection and shadow rays. To maintain BVH quality, some parts of the tree have to be rebuilt. BART (shown in Fig. 4) is a part of the BART animated ray tracing benchmark [16] and shows a set of triangles with mostly unstructured, random movement. Since it has high depth complexity and overlapping primitives, this scene is one of the worst cases for BVH rendering as well as hierarchy updates. For the former, we have found that the ordering approach for BVHs ameliorates the effects of depth complexity. Additionally, the independent movement of each triangle leads to extreme degradation in BVH quality, so that our heuristic rebuilds the tree quite often. Finally, we tested a more complex static scene of the 1M triangle Stanford Buddha (not shown) to demonstrate that BVH ray tracing can compete with kd-trees even for larger models. Unfortunately, the update time grows linearly with model size, so a more efficient update scheme would be needed to be able to render this or any larger model at high frame rates.

We tested our heuristic for tree rebuilding on the test models and found that in all cases except the BART model, just hierarchy updates can be efficient enough for rendering. The unstructured, random movement of triangles in the BART scene makes several tree rebuilds necessary, however. Without doing that, we found that frame rates will decrease by over an order of magnitude in just a few frames. To test how well the rebuild times are chosen, we benchmarked the animation while rebuilding only via heuristic (with the threshold set to 0.4) as well as rebuilding the hierarchy every frame.

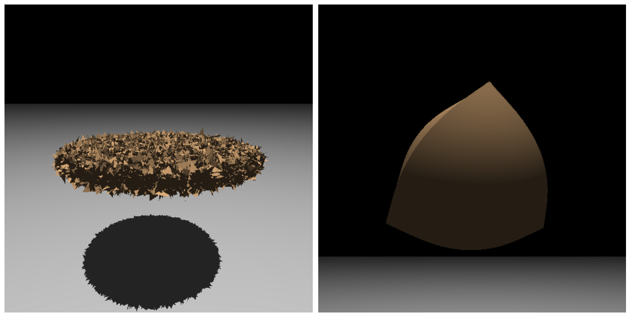

Figure 4: **BART Museum triangles:** *Two image shots from* 170 *steps of a randomly deforming model from the BART deforming data benchmark. We achieve an average of* 11 *frames per second at* 512^2 *screen resolution using a dual-core P4 processor at 2.8 GHz.*

Scene	Tris	build	update time/frame	avg. fps
Bunny/Cloth	16K	13 ms	4ms	13
BART model	16K	23 ms	6ms	11
Dress model	40K	41 ms	14ms	13
Bunny	69K	90 ms	23ms	6
Buddha	1M	1659 ms	220ms	3

Table 1: **Benchmarks and Timings:** *Results for BVH ray tracing of several scenes. The benchmark configuration for each of the scenes is described in section 5. All benchmarks were performed at* 512^2 *resolution on a dual-core P4 machine at 2.8 GHz using 8x8 ray packet traversal and secondary rays (shadows and reflections). Performance numbers are given as an average over the whole animation, tree build times are for the spatial median build.*

We found that even when looking just at pure rendering time without counting rebuilding and updating, the animation rendered with new hierarchy in each frame was only 20% faster than rendering using our heuristic. The latter needed only a few rebuilds, so the total overhead incurred by updates and rebuilds was only 2s over the whole sequence, as compared to 15s for rebuilding.

6 Comparison to spatial kd-trees

Recently, several acceleration structures were proposed that could be seen as a hybrid between BVHs and kd-trees. The basic idea is that when looking at BVH construction, it is apparent that storing full bounding boxes may be redundant if a node essentially just stores the geometry as split along one dimension. Unlike kd-trees, which solve this by storing just the actual split coordinate and dimension, *spatial kd-trees* for ray tracing [10, 37] store two coordinates which represent the limits of the bounding boxes of the left and right child in the split dimension (which are allowed to overlap in case the contained geometry does). This reduces the memory requirements from storing 6 to 2 coordinates only. Similarly, Woop *et al.* [40] present a hardware implementation called the *b-kd-tree* in which they store the left and right bounds for both children and therefore use 4 coordinates per node. Construction for both structures is almost identical to BVHs by just storing the respective bounding box coordinates. Both approaches also decrease the actual work done for one intersection as only 2 or 4 planes need to be intersected against the ray. In general, the traversal algorithm for spatial kd-trees is more similar to kd-trees with the difference that rays are now intersected against multiple planes. However, unlike kd-trees, no real depth sorting is provided, so traversal cannot stop after the first hit.

Implementation: To compare our BVH implementation against those approaches, we implemented a spatial kd-tree structure with

		BVH:			kd-tree:			spatial kd-tree:			
Scene	Tris	nodes	memory	build	nodes	memory	build	nodes	memory	build	update
Bunny/Cloth	16K	31923	997 KB	170 ms	64137	859 KB	1487ms	35097	548 KB	146ms	4ms
BART model	16K	32767	1024 KB	322 ms	11075	1426 KB	1902ms	58921	920 KB	331ms	11ms
Dress model	40K	80059	2501 KB	733 ms	218845	2778 KB	5s	148929	2327 KB	821ms	19ms
Bunny	69K	138901	4340 KB	1526 ms	442347	5072 KB	10s	259543	4055 KB	1521ms	37ms
Buddha	1M	2175431	67982 KB	32s	2989439	33225 KB	80s	3666989	57296 KB	31s	490ms

Table 2: **BVH compared to kd-tree and skd-tree:** *Tree statistics for other acceleration structures as compared to BVHs. All hierarchies were built using the surface area heuristic instead of the spatial median and BVH build times are therefore higher than in the previous table (using the same machine). The SAH construction uses the simple $O(n\log^2 n)$ algorithm as opposed to the faster $O(n\log n)$ version [30]. Note that the memory requirements for skd-trees are only slightly smaller than for BVHs due to the higher number of nodes. Build times are about the same for both.*

	1 ray		2×2		8×8		16×16	
Scene	BVH	skd-tree	BVH	skd-tree	BVH	skd-tree	BVH	skd-tree
Bunny/Cloth	1.8	3.1	3.5	7.1	7.7	9.0	7.9	8.6
BART model	0.9	0.9	2.9	4.7	7.5	8.5	8.1	8.2
Dress model	1.3	2.3	3.3	5.7	6.9	7.9	7.1	6.7
Bunny	1.1	1.7	2.6	4.3	5.6	6.2	5.6	5.6
Buddha	1.0	1.3	1.7	2.7	3.0	2.9	2.2	1.5

Table 3: **Rendering performance of BVH and spatial kd-tree:** *Direct comparison of rendering times for BVH and our spatial kd-trees implementation. Benchmark results are average frames per second over the animation for 1024^2 primary rays only on a dual-core P4 machine at 2.8 GHz. The results are shown for different ray packet sizes and exclude all update times and rebuild times. In order to avoid excessive quality degradation when updating skd-trees, we rebuild the hierarchy every 5 frames.*

two planes as in [37]. Similar to BVHs, we use 2×2 ray packets using SIMD instructions as well as packets of arbitrary size to allow direct comparison of results. For 2×2 packets, the traversal algorithm is a direct adaptation of ray packet traversal in kd-trees extended to test two planes. For larger packets, we designed a traversal algorithm that uses the inverse frustum culling described for kd-tree ray tracing in [22] for determining whether a packet intersects a node, although we do not perform entry-point search or split packets at the moment.

Results and discussion: Our results are summarized in Table 3. In general, we observed an increase in overall rendering speed for static scenes when using skd-trees, which is a consequence of the less computationally expensive ray-node intersection. However, for animations one important disadvantage is that after the update, many of the empty leaf nodes introduced for empty space elimination may not be necessary any more or, even worse, would have to be used at a different point. We have found that this quickly results in more severe quality degradation of the hierarchy and, subsequently, rendering requires very frequent rebuilds. To avoid this, we rebuilt the hierarchy every 5 frames for skd-trees, to allow a better performance comparison.

We also found that traversal for larger ray packets does not seem to scale up as well as for BVHs, so that skd-trees are about the same speed or even slower for our tested packet sizes. Although the individual nodes are only half as large as our AABB nodes (i.e. 16 bytes), memory use for skd-trees is only slightly lower than for BVHs. The reason for this is that in order to achieve good performance, extra splits to eliminate empty space at the outer bounds are needed often and, even though the actual empty leaves do not need to be stored, this increases the number of actual nodes in the tree. This also means that unlike BVHs, the actual number of nodes for a scene is not as predictable (although it can be bounded easily since only a limited number of empty space subdivisions can be introduced at each node), which prevents some easy ways to optimize construction. Having more nodes also means that the tree is deeper and therefore on average more traversal steps may be needed to reach the leaves. Most importantly, though, the hierarchy update for animation is linear to the number of nodes. As the skd-tree usually has about twice as many nodes, this means that updating it will also take twice as long, which may become the bottleneck in an interactive application.

Finally, a subtle difference is that ray packet traversal for skd-trees in general can be more complicated to implement and less versatile: as for kd-trees, groups of rays for inverse frustum culling are limited to having the same direction signs, which can make obtaining groups of coherent rays more challenging, in particular for secondary rays. In contrast, BVH traversal is independent of ray direction signs, which eliminates special cases for traversal, and frustum culling can be introduced easily by the fast frustum-box intersection described in [22].

This leads us to conclude that for animated scenes with updates a BVH implementation is to be preferred and also lends itself better to an optimized ray packet implementation. For scenes with varying numbers of primitives, the hierarchy update will not work, so in that case a fast rebuild such as described in [37] should be used instead. For static scenes, standard kd-trees will very likely provide superior performance with an MLRT implementation [22], albeit at higher memory cost and more complex optimized hierarchy construction.

7 Future Work and Conclusion

We have proposed an algorithm for interactive ray tracing of deformable, animated models. We used BVH hierarchies as an acceleration data structure of the deformable models and showed optimizations that will result in performance competitive or even exceeding rendering using kd-trees. We were also able to integrate efficient ray coherence techniques for kd-trees to our BVHs. We do not make any assumptions about the possible deformation or motion of objects and dynamically update or rebuild the hierarchy depending on our simple heuristic.

There are many interesting directions for future work. Our current algorithm is mainly designed for small to intermediate model complexity. We would like to extend our algorithm to handle larger deforming models, which would require more efficient or localized update methods that only change the parts of the hierarchy that have deformed since the last frame. Also, we would like to investigate cache-coherent layout computation methods [42–44] of deforming models in order to efficiently handle them. Another interesting problem is the better use of multiprocessor architectures in the context of hierarchy construction and updates. We plan to extend our current methods to be more general and flexible for these applications. Finally, we think that it would be interesting to improve the simple construction method we use to effiently approximate instead of fully computing the surface area heuristic, thus allowing better performance without adding too much overhead.

8 Acknowledgments

Part of this work was performed while the primary author was working at Intel corporation. We would like to thank Ingo Wald, Alexander Reshetov, Gordon Stoll and UNC's GAMMA group for helpful discussions and feedback, as well as the anonymous reviewers for their comments. The model used for the dress simulation is courtesy of Walt Disney Feature Animation and the Bunny and Buddha models are courtesy of the 3D scanning repository at Stanford University. This work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, ONR Contract N00014-01-1-0496, DARPA/RDECOM Contract N61339-04-C-0043, Intel, and LOCAL LLNL LDRD project (05-ERD-018) and was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

References

[1] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968.

[2] Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Interactive Distribution Ray Tracing. *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-022*, 2006.

[3] Per H. Christensen, David M. Laur, Julia Fong, Wayne L. Wooten, and Dana Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *j-CGF*, 22(3):543–552, September 2003.

[4] David Cline, Kevin Steele, and Parris K. Egbert. Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools: JGT (to appear)*, 2006.

[5] Markus Geimer and Stefan Müller. A Cross-Platform Framework for Interactive Ray Tracing. In *Graphiktag im Rahmen der GI Jahrestagung*, Frankfurt am Main, 2003.

[6] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987.

[7] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, pages 171–180, 1996.

[8] Johannes Günther, Heiko Friedrich, Ingo Wald, Hans-Peter Seidel, and Philipp Slusallek. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3), September 2006. to appear.

[9] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[10] Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. On Fast Construction of Spatial Hierarchies for Ray Tracing. *Submitted to RT'06*, 2006.

[11] P. M. Hubbard. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.

[12] Homan Igehy. Tracing ray differentials. In *ACM SIGGRAPH*, pages 179–186, 1999.

[13] J. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics*, 4(1):21–37, 1998.

[14] T. Larsson and T. Akenine-Möller. Strategies for bounding volume hierarchy updates for ray tracing of deformable models. Technical report, 2003.

[15] Thomas Larsson and Tomas Akenine-Möller. Collision detection for continuously deforming bodies. In *Eurographics*, pages 325–333, 2001.

[16] J. Lext, U. Assarsson, and Tomas Akenine-Möller. A benchmark for animated ray tracing. In *IEEE Computer Graphics and Applications*, 2001.

[17] Jonas Lext and Tomas Akenine-Möller. Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001, short presentation*, 2001.

[18] Jeffrey Mahovsky. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, September 2005.

[19] Jeffrey Mahovsky and Brian Wyvill. Fast ray-axis aligned bounding box overlap tests with plücker coordinates. *journal of graphics tools*, 9(1):35–46, 2004.

[20] Steven G. Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian E. Smits, and Charles D. Hansen. Interactive ray tracing. In *SI3D*, pages 119–126, 1999.

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

[22] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005.

[23] S. M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[24] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. AK Peters Limited, second edition, 2003.

[25] Peter Shirley, Philipp Slusallek, Bill Mark, Gordon Stoll, and Ingo Wald. Introduction to real-time ray tracing. *SIGGRAPH Course Notes*, 2005.

[26] Brian Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools: JGT*, 3(2):1–14, 1998.

[27] Abe Stephens, Solomon Boulos, James Bigler, Ingo Wald, and Steven Parker. An application of scalable massive model interaction using shared memory systems. *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization (to appear)*, 2006.

[28] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino. Collision detection for deformable objects. *Computer Graphics Forum*, 19(1):61–81, 2005.

[29] G. van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, 2(4):1–14, 1997.

[30] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). SCI Institute Technical Report UUSCI-2006-009, University of Utah, 2006.

[31] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[32] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[33] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In Alan Chalmers and Theresa-Marie Rhyne, editors, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)*, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001.

[34] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *Technical Report, SCI Institute, University of Utah, No UUSCI-2005-014 (conditionally accepted at ACM Transactions on Graphics)*, 2006.

[35] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proceedings of the Eurographics Symposium on Ren-*

dering, 2004. (to appear).

[36] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *Technical Report, SCI Institute, University of Utah, No UUSCI-2005-014 (conditionally accepted at ACM SIGGRAPH 2006)*, 2006.

[37] Carsten Wächter and Andreas Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006: Eurographics Symposium on Rendering.*, 2006.

[38] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

[39] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools: JGT*, 10(1):49–54, 2005.

[40] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware (to appear)*, 2006.

[41] Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. R-LODs: Interactive LOD-based Ray Tracing of Massive Models. *The Visual Computer (Pacific Graphics)*, 2006. To appear.

[42] Sung-Eui Yoon and Peter Lindstrom. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2006)*, 12(5), September-October 2006.

[43] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-Oblivious Mesh Layouts. *Proc. of ACM SIGGRAPH*, 2005.

[44] Sung-Eui Yoon and Dinesh Manocha. Cache-Efficient Layouts of Bounding Volume Hierarchies. In *Computer Graphics Forum (Eurographics)*, 2006. To appear.