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

Christian Lauterbach, Sung-Eui Yoon, David Tuft, Dinesh Manocha

University of North Carolina at Chapel Hill

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. Our algorithm has been applied to different scenarios composed of tens of thousands to a million triangles arising in animation and simulation. In practice, our algorithm can ray trace these models at 4-20 frames a second on a dual-core Xeon PC.

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. Over 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 [RSH05, Wal04]. 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 many seconds even on models composed of tens of thousands of triangles [Hav00, WH06]. 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. 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 secondary and shadow rays at $4-20$ frames per second on a dual-core 2.8GHz Xeon PC with 2GB of memory.

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.

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. Finally, we show the results obtained by our approach on several benchmarks in section 5.

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 [App68, Whi80], the ray tracing algorithm has been very well studied in computer graphics due to its generality and high rendering quality. Recently, several systems have been presented that are capable of generating ray traced images at interactive speeds. A recent survey is given in [SSM*05]. Parker *et al.* [PMS*99] present a real-time ray tracing algorithm on a shared-memory supercomputer. Several approaches use ray coherence to improve performance and achieve interactive

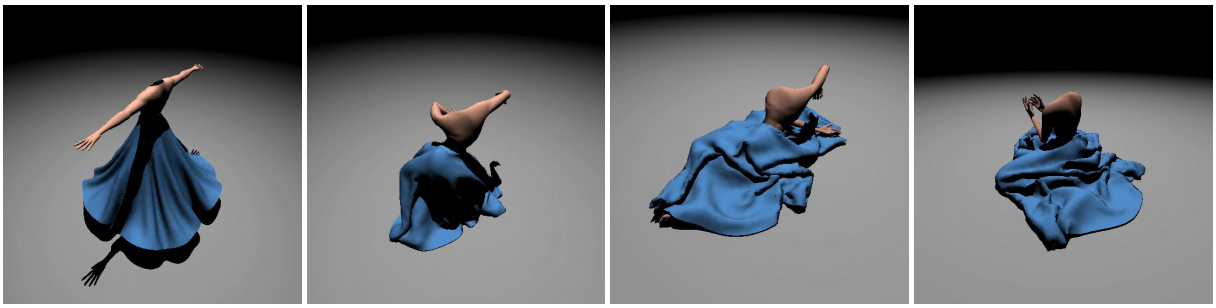

Figure 1: ***Princess Model****: Four different images of a* 220 *step sequence from a dynamic cloth simulation consisting of* 40*K triangles. By computing and updating the AABB hierarchy of the deforming model, we are able to achieve* 16 *frames per second on dual Xeon processors.*

performance on commodity desktop systems for large static datasets, such as coherent rat tracing [WBWS01, Wal04]. More recently, MLRT [RSH05] integrates kd-tree traversal with beam tracing to further improve performance.

Dynamic Scenes: There is relatively less work on ray tracing dynamic scenes. Reinhard et al. [RSH00] use a grid structure that can be updated efficiently for any type of animation. Lext *et al.* [LAAM01] 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 [LAM01b]. Wald et al. [WBS03] describe a distributed system for dynamic scenes that differentiates between transformations and unstructured movement in the scene. Recently, Ingo *et al.* [WIK*06] proposed coherent grid traversal algorithm to handle dynamic models.

Bounding volume hierarchies: BVHs have been widely used to accelerate the performance of ray tracing algorithms [RW80, Smi98]. In the case of static scenes, algorithms based on kd-trees and nested grids seem to outperform BVH-based algorithms [Hav00]. Larsson and Akenine-Möller [LAM01a] 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 [LAM03]. BVHs have also been used to accelerate the performance of collision detection algorithms for deformable models [vdB97, TKH*05].

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 [Hub93], OBB-trees [GLM96], and k-DOP-trees [KHM*98]. However, we use AABBs as the BV as they provide a good balance between the tightness of fit and computation cost. We also use efficient algorithms for ray-box intersection [SM03, WBMS05].

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 [Wal04]. As far as static scenes are concerned, analysis has shown that optimized algorithms based on kd-trees will outperform BVH-based algorithms [Hav00]. There are multiple reasons to explain this behavior: First, even the most optimized ray-AABB intersection test (e.g. from [WBMS05]) 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 if 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 [WDS04]) 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

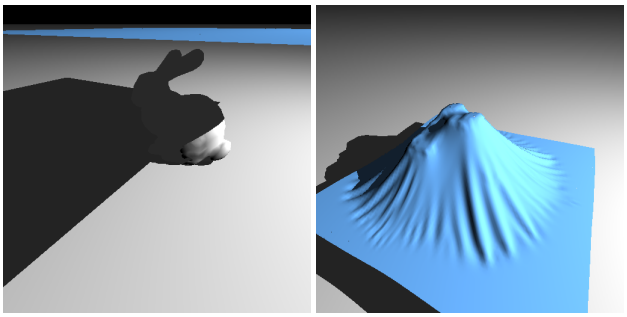

Figure 2: ***Cloth on Bunny Simulation:*** *Two shots of a* 315 *step dynamic simulation of cloth dropping on the Stanford bunny. We achieve* 19 *frames per second on average during ray tracing of this deforming model.*

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 to primitive intersection. During hierarchy construction, the most important operation is to find a divider for the two subsets that will 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) [GS87], which has been shown to yield higher ray tracing performance. However, it also has a much higher construction cost, which can take a significant fraction of a frame time for dynamic environments. Because of this, we use the midpoint 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 midpoint heuristic provides good rendering performance and is very fast to compute. 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 recursive AABB hierarchy construction algorithm, given an input model consisting of n primitives, satisfies $T(n) = kT(\frac{n}{k}) + O(n)$ due to its recursive formulation, where k is the number of children of each node. Therefore, the time complexity is $O(n\log_k(n))$.

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 fast update times (see Table 1), which can be one order of magnitude lower than rebuilding the tree for models with hundreds of thousands of polygons. 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 reclustering 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 and 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. 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 as 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 between 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 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.

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 or point lights) any single 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. Another issue is cache coherence during traversal: similar to the compact kd-tree representations [WDS04], we can optimize the AABB representation to fit within 32 bytes of data, which also includes the information that is needed to rebuild the tree. Our profiling shows that BVH traversal using our AABBs has the same cache efficiency as the kd-tree traversal.

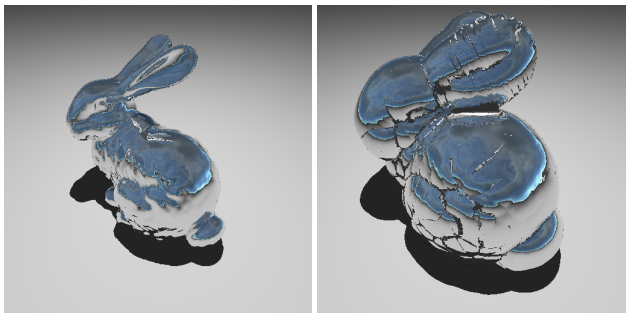

Figure 3: ***Bunny blowing up :*** *Two images show frames from a* 113 *step animation of a deforming Stanford bunny. We achieve* 8 *frames per second on average during ray tracing this deforming model with shadow and reflection rays.*

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 [WBWS01] 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 2x2 ray bundles and found that it yields a speedup of about 2 to 3, which is even above the improvement obtained for kd-trees. Adapting the MLRT algorithm [RSH05] to BVHs is also straightforward.

4.2. Multi-Core Architectures

One of major features of current computing trends is that there are multi-cores and hyper-threading functionality available on commodity architectures. Therefore, it is desirable to design

our hierarchy construction, update, and runtime traversal such that they take advantage of available parallelism.

Hierarchy construction: There are no good and optimal algorithms that can easily parallelize hierarchy construction. Since ray tracing scales well with multiple processors, it is desirable to speed up construction by distributing the work over several threads and cores. To achieve this, we first divide a set of triangles and vertices up to four sets by using one thread. Then, we assign each thread to construct a sub-BVH for each divided set. In general, this may not achieve high load balancing. However, we found that this simple method works well with our benchmarks since BVHs of our benchmarks are well balanced.

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. Table 1 shows the timings for our results. 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. We found that this approach works well with our benchmarks.

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 Intel Xeon 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). Both acceleration structures support ray packet traversal using the SSE SIMD instruction set on Intel processors. For efficiency reasons, we only support triangles as primitives. To speed up rendering, 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 Table 1). In general, building a BVH tree using the naive midpoint method is much 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 primitives can be located in multiple nodes.

Benchmarks: We show five different test cases (Refer Table 1): Scene 1 (shown in Fig. 2) and Scene 3 (shown in Fig. 1) 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. Scene 4 (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. Scene 2 (shown in Fig. 4) is a part of the BART animated ray tracing benchmark [LAAM01] 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 update. 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 parts of the tree quite often. Finally, we demonstrate a more complex static scene of 1M Buddha (Scene 5) to show 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. 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. 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 algo-

Scene	Triangles	**BVH:**nodes	memory	build time	update time	fps
1) Cloth on bunny	16K	31923	997 KB	98 ms	4ms	19
2) BART model	16K	32767	1024 KB	96 ms	5ms	12
3) Cloth model	40K	80059	2501 KB	224 ms	8ms	16
4) Bunny	69K	138901	4340 KB	395 ms	11ms	8
5) Buddha	1M	2175431	67982 KB	7593 ms	167ms	4

Scene	Triangles	**kd-tree:**nodes	memory	build time
1) Cloth on bunny	16K	64137	859 KB	1487ms
2) BART model	16K	11075	1426 KB	1902ms
3) Cloth model	40K	218845	2778 KB	5s
4) Bunny	69K	442347	5072 KB	10s
5) Buddha	1M	2989439	33225 KB	80s

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. The top table shows the performance for a BVH. The bottom table shows the tree computation time and memory overhead for a kd-tree of the same model (for comparison). All benchmarks were performed at* 512^2 *resolution on a dual Xeon machine at 2.8 GHz using 2x2 ray packet traversal. Both hierarchies were built using a single thread only.*

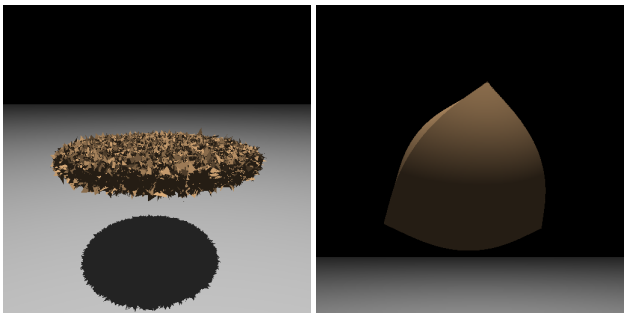

Figure 4: ***BART Randomly Moving Triangles:*** *Two image shots from* 170 *steps of a randomly deforming model from the BART deforming data benchmark. We are able to achieve* 12 *frames per second on average during ray tracing this model with shadow rays.*

rithm to handle larger deforming models, which would require more efficient or localized update methods. 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. We found that there is concurrent work on ray tracing of dynamic models based BVHs [WBS06] with ours. We would like to perform a detailed comparison of our algorithm with theirs.

References

[App68] APPEL A.: Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.* (1968), vol. 32, pp. 37–45.

[GLM96] GOTTSCHALK S., LIN M., MANOCHA D.: OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96* (1996), 171–180.

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7*, 5 (1987), 14–20.

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[Hub93] HUBBARD P. M.: Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality* (October 1993).

[KHM*98] KLOSOWSKI J., HELD M., MITCHELL J., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics 4*, 1 (1998), 21–37.

[LAAM01] LEXT J., ASSARSSON U., AKENINE-MÖLLER T.: A benchmark for animated ray tracing. In *IEEE Computer Graphics and Applications* (2001).

[LAM01a] LARSSON T., AKENINE-MÖLLER T.: Collision detection for continuously deforming bodies. In *Eurographics* (2001), pp. 325–333.

[LAM01b] LEXT J., AKENINE-MÖLLER T.: Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001, short presentation* (2001).

[LAM03] LARSSON T., AKENINE-MÖLLER T.: *Strategies for Bounding Volume Hierarchy Updates for Ray Tracing of Deformable Models*. Tech. rep., 2003.

[PMS*99] PARKER S. G., MARTIN W., SLOAN P.-P. J., SHIRLEY P., SMITS B. E., HANSEN C. D.: Interactive ray tracing. In *SI3D* (1999), pp. 119–126.

[RSH00] REINHARD E., SMITS B., HANSEN C.: Dynamic acceleration structures for interactive ray tracing. In *Proceedings Eurographics Workshop on Rendering* (June 2000), pp. 299–306.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3 (2005), 1176–1185.

[RW80] RUBIN S. M., WHITTED T.: A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics 14*, 3 (July 1980), 110–116.

[SM03] SHIRELY P., MORLEY R. K.: *Realistic Ray Tracing*, second ed. AK Peters Limited, 2003.

[Smi98] SMITS B.: Efficiency issues for ray tracing. *Journal of Graphics Tools: JGT 3*, 2 (1998), 1–14.

[SSM*05] SHIRLEY P., SLUSALLEK P., MARK B., STOLL G., WALD I.: Introduction to real-time ray tracing. *SIGGRAPH Course Notes* (2005).

[TKH*05] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FÜHRMANN A., CANI M.-P., FAURE F., MAGNENAT-THALMANN N., STRASSER W., VOLINO P.: Collision detection for deformable objects. *Computer Graphics Forum 19*, 1 (2005), 61–81.

[vdB97] VAN DEN BERGEN G.: Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools 2*, 4 (1997), 1–14.

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

[WBMS05] WILLIAMS A., BARRUS S., MORLEY R. K., SHIRLEY P.: An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools: JGT 10*, 1 (2005), 49–54.

[WBS03] WALD I., BENTHIN C., SLUSALLEK P.: Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)* (2003).

[WBS06] WALD I., BOULOS S., SHIRLEY P.: 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).

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)* (2001), Chalmers A., Rhyne T.-M., (Eds.), vol. 20, Blackwell Publishers, Oxford, pp. 153–164.

[WDS04] WALD I., DIETRICH A., SLUSALLEK P.: An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proceedings of the Eurographics Symposium on Rendering* (2004). (to appear).

[WH06] WALD I., HAVRAN V.: *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.

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Commun. ACM 23*, 6 (1980), 343–349.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: 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).