Fast, Effective BVH Updates for Dynamic Ray-Traced Scenes Using Tree Rotations

Daniel Kopta Andrew Kensler Thiago Ize Josef Spjut Erik Brunvand Al Davis

Abstract

Bounding volume hierarchies are a popular choice for ray tracing animated scenes due to the relative simplicity of refitting bounding volumes around moving geometry. However, the quality of such a refitted tree can degrade rapidly if objects in the scene deform or rearrange significantly as the animation progresses, resulting in dramatic increases in rendering times. Existing solutions involve occasional or heuristically triggered rebuilds of the BVH to reduce this effect. In this work, we describe how to efficiently extend refitting with local restructuring operations called tree rotations which can mitigate the effects that moving primitives have on BVH quality by rearranging nodes in the tree during each refit rather than triggering a full rebuild. The result is a fast, lightweight, incremental update algorithm that requires negligible memory, has minor update times and parallelizes easily, yet avoids significant degradation in tree quality or the need for rebuilding while maintaining fast rendering times. We show that our method approaches or exceeds the frame rates of other techniques and is consistently among the best options regardless of the animation scene.

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

Keywords: ray tracing, acceleration structures, bounding volume hierarchies, tree rotations, dynamic scenes, parallel update

1 Introduction and Background

Acceleration structure maintenance is a crucial component in any interactive ray tracing system with dynamic scenes. As the geometry changes from frame to frame, the existing acceleration structure must be either updated or replaced with a new one, the latter of which can be costly. An ideal update algorithm should produce an acceleration structure that is as efficient to render as one rebuilt from scratch by a high quality, offline build algorithm for that same frame, yet it should produce it in as little time as possible.

In recent years, bounding volume hierarchies (BVHs) have been a popular subject for research on efficient acceleration structure update algorithms. They are also popular for our specific area of interest which is animations where geometry is moved or deformed on each frame. BVHs are relatively fast to render, and there is a very simple update algorithm that involves node refitting [Wald et al. 2007] to handle moving and deforming geometry. Refitting works by performing a post-order traversal of the nodes in the BVH tree. Each leaf is updated with a new tight bounding volume around its corresponding geometry, and the interior nodes combine these to form a tight volume enclosing their children. With axis-aligned bounding boxes, this process is fast and reasonably effective for small deformations to the underlying geometry. However, the quality of the tree can degrade rapidly when the geometry moves incoherently or undergoes large topological changes.

Full rebuild algorithms for animations overcome this by replacing the BVH with a new one before the degradation becomes too significant. [Lauterbach et al. 2006] measures the degradation and performs a rebuild on demand; while this is able to maintain low average frame times, it introduces noticeable rendering pauses when a new tree must be built.

[Ize et al. 2007] perform the rebuild asynchronously while refitting frame to frame so that the tree quality never has a chance to become too poor and rendering stalls never occur as with the method of Lauterbach et al. While this removes the rendering stalls produced by the method of Lauterbach et al., the asynchronous rebuild requires a dedicated core just for rebuilding, significant changes to the ray tracing system, adds an additional one frame lag in user input, and requires storing two copies of the mesh and BVH.

[Wald 2007] avoids degradation by using a fast parallel build algorithm to completely rebuild the tree for every frame. This has the advantage of supporting all types of animations in addition to the deformations handled by refitting, and allows each frame to have an un-degraded tree. However, in order to achieve very low build times, Wald had to produce lower quality trees than those used in a high quality sweep or binned SAH. Furthermore, even these very fast parallel builds were still significantly slower than refitting and did not scale well to many cores, which makes per frame rebuilds attractive only for animations with significant deformation or small triangle counts.

[Wald et al. 2008] later combined fast parallel rebuilds with the asynchronous rebuild in order to ensure that every few frames a new tree would be available. This resulted in fairly good performance for animations with significant deformation; however it still has the disadvantage that between new trees performance can still significantly degrade and for animations with less deformation the overhead of dedicating multiple cores solely to building, in addition to the refitting, results in inferior performance compared to just refitting.

Hybrid algorithms combine refitting with heuristics to determine when to perform a partial rebuild or restructuring of a sub-tree. [Yoon et al. 2007] uses a cost/benefit estimate of the culling efficiency of ray intersection tests to restructure pairs of nodes, while [Garanzha 2008] looks for nodes whose children undergo divergent motion. Both of these algorithms use multiple phases to first identify candidates for restructuring, and then reconstruct them.

On a GPU, [Lauterbach et al. 2009] showed how a BVH could be quickly built using the LBVH algorithm. However, tree quality was significantly inferior to that produced by a standard SAH build, especially if the model did not consist of uniformly distributed triangles. [Pantaleoni and Luebke 2010] improved on the LBVH build time and tree quality by exploiting spatial coherence in the original mesh and performing a SAH build over the top level of the tree. However, their HLBVH algorithm still depends on the LBVH

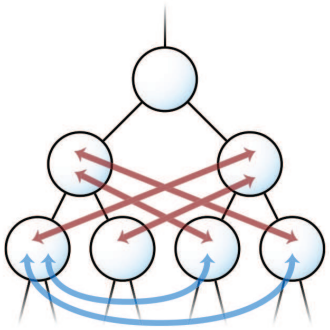


Figure 1: *Potential tree rotations considered*

and so suffers with nonuniform triangle distributions. Though a very fast build, it is still two orders of magnitude slower than just refitting. [Garanzha et al. 2011] further improves on HLBVH using work queues, and is able to reduce the build time further, but is still almost an order of magnitude slower than refitting alone. These algorithms, however, are targeted specifically to GPUs and have not been shown to be effective update strategies for a CPU-based build.

1.1 Tree Rotations for BVHs

Tree rotations for bounding volume hierarchies were first introduced in [Kensler 2008], in which rotations are applied as a preprocess on top of an offline build algorithm to improve the quality of the BVH for static scenes. The algorithm starts with a BVH built from a greedy surface area heuristic (SAH) [Goldsmith and Salmon 1987] construction, it then considers potential improvements to the tree via restructuring operations called tree rotations. Making hundreds of full passes over the tree, the algorithm is able to reduce the SAH cost and render time by up to 18% for static scenes.

Figure 1 shows the potential node swaps that the algorithm considers. Each of the upper four rotations are the base primitive rotations and involve exchanging a direct child of the node with a grandchild on the opposite side. This has the effect of raising one subtree at the expense of lowering the other. The lower two rotations are compound rotations that can be composed through a sequence of the upper four. These compound rotations are used in order to give the algorithm the ability to find improvements that it might miss due to the intermediate steps raising the SAH cost temporarily, and thus getting stuck in local minima. Note that the figure is not symmetrical with respect to the lower rotations on the grandchildren because the missing two rotations merely produce mirrored trees. Since a tree and its mirror share the same cost, that would result in redundant work. This is the basic hill-climbing algorithm of [Kensler 2008].

Since the BVH is built from top down, it can only make estimates about the true costs of the subtrees it is building. Once the tree is built, however, we can know the true SAH cost of any given subtree. Kensler's algorithm uses this knowledge to consider swapping certain nodes to place them under a different parent that can bound them more efficiently as measured by the true SAH value. After one pass over the tree, its new configuration presents new swapping opportunities that were not apparent on the first pass. Any given subtree may need to undergo a sequence of multiple rotations to reach the optimal configuration. Since the intermediate steps to reach that state may temporarily have a higher SAH cost, a simple hill climbing approach is not sufficient. To solve this problem, simulated annealing [Kirkpatrick et al. 1983] is applied, which allows for detrimental changes to happen occasionally in order to avoid local minima. This requires that many hundreds of passes over the tree are made in order to eventually converge on a high quality solution.

Since the algorithm must make so many passes over the tree, it can add several minutes to the build time, and up to 14 minutes for the largest model tested. Using a high quality sweeping SAH build algorithm, this translates to about 2 orders of magnitude increase in build time. Scenes with uniformly distributed and sized triangles such as laser scanned models tend to already be near the optimum configuration from the initial build, so tree rotations only marginally improve the SAH cost, and can even result in a slight increase in render time, since the correlation between SAH cost and render time is not strictly related. Scenes with heterogeneous triangle distribution and sizes such as architectural scenes see greater improvements. In order to reach the reported 18% performance improvement, however, the renderer must employ path tracing and not just simple ray casting. This is likely due to the assumption that the SAH makes about rays being evenly distributed in origin and direction [Kensler 2008].

2 Incremental Updates via Tree Rotations

In this paper, we take the tree rotations as described by [Kensler 2008], which was a slow offline preprocess used on high quality trees in order to get slightly higher quality trees, and instead make the observation that while it is challenging to improve upon an already high quality tree, tree rotations have a much easier time improving upon a lower quality tree and the iterations can be applied across successive frames even if the geometry is deforming across those frames. This is precisely what occurs during an animation using refitting, especially if geometry is moving incoherently. By folding tree rotations into the refitting operations, we are able to efficiently use tree rotations to dramatically improve the quality of the trees generated by a BVH refit with only a small increase in additional processing time and so can achieve better performance on animations than refitting alone or per frame parallel rebuilds.

On each frame, after the geometry has been interpolated to its new position, we do a standard refit which involves a post-order traversal of the tree, during which we refit each node to enclose the underlying geometry of its two child nodes. After refitting the node, the subtree rooted at that node may no longer be structured in a manner that is efficient for ray traversal. As triangles move apart from one another, the tree structure that was built assuming their previous positions can have awkward relationships between sibling nodes. While the BVH has not changed structurally, the volumes within it have changed spatially, potentially resulting in significant node overlap, making it difficult to efficiently cull off sections of the tree.

After refitting a node using the traditional refitting algorithm, we now consider restructuring its direct children and grandchildren with tree rotations. This allows for two nodes to swap positions in the tree, giving them a different parent and grouping them with a different sibling. If this swap results in a better spatial organization for the subtree, as represented by its SAH cost, then it is considered a beneficial reconfiguration and the swap is kept. If a swap is made, we then update the bounding volume of the affected parent node with another refit operation that tightly bounds its new children. As the tree quality degrades from moving geometry and refitting, our tree rotations are able to find even more useful swaps to perform, and can maintain the tree in a high quality state. Figure 2 illustrates this process.

The node that we are considering rotations for is the outer black bounding node that contains all 3 triangles in Figure 2. The green triangle (b) that moves from one frame to the next was originally grouped in a subtree with triangle (a) as its sibling. After (b) moves, their parent node is refit, shown in red. This new red bounding volume does not efficiently contain its two children, and results in a large empty space and higher SAH cost. Rays passing through that

empty space ideally would not need to test against either of the two nodes contained within that subtree, so a better tree organization may be possible. After checking for beneficial rotations, we find that swapping the leaf node containing triangle (a) with the leaf node containing triangle (c) produces a tree with a lower SAH cost, and the new red node contains less empty space. This process is done from the bottom up for every node in the tree that has grandchildren.

Since Kensler's hill climbing algorithm for tree rotations can be implemented with a post-order traversal and works on the same data as visited during refitting, we can remove most of the memory access costs incurred by tree rotations by performing both the refitting and rotations in the same pass over the tree. This results in a fast update algorithm that is easy to add to any ray tracing system that already uses refitting, and maintains a high quality tree without the need for rebuilding. Furthermore, if the refitting is already parallelized, then the rotations will also be parallelized simply by being added into the refit.

The biggest caveat of our algorithm involves primitives and leaf nodes during construction of the original tree. Primitives whose motion may diverge need to be placed in separate leaf nodes in order to prevent large empty spaces forming in their parent bounding volume. In practice, this means that our implementation builds the tree down to the level of a single primitive per leaf node, leading to potentially larger trees than those produced by the SAH. A smarter construction algorithm with a priori knowledge of how primitives move together [Günther et al. 2006] could improve on this. Another option would be to split and merge leaf nodes during rotation.

3 Results

To evaluate the performance of our algorithm, we implemented it in the Manta interactive ray tracer [Bigler et al. 2006]. Starting with the existing recursive refitting code, we extended this to also maintain cost evaluations and to perform the most beneficial tree rotation as each node is visited on each frame. For benchmarking purposes, all results were gathered on a 2.67GHz 8-core Intel Xeon X5550 running Manta, rendering all scenes at 1024x1024 pixels with shadows for a single point light source. All refitting and rotation updates and renderings used 8 parallel threads. For BVH traversal, we use 8x8 ray packets and the interval arithmetic culling scheme described in [Wald et al. 2007].

Our example animations, seen in Figure 3, fall into two broad categories: smaller scenes with simple deformations (Cloth Ball (92k triangles) and Fairy Forest (174k)), and scenes with a variety of sizes, but with extensive deformations (Exploding Dragon and Bunny (253k), Lion (1600k), N-Body Simulation (146k), and BART Museum (66k)). We measured the performance on each animation using refitting only, and our proposed refitting with rotations. We also measured performance on two baseline techniques: performing a

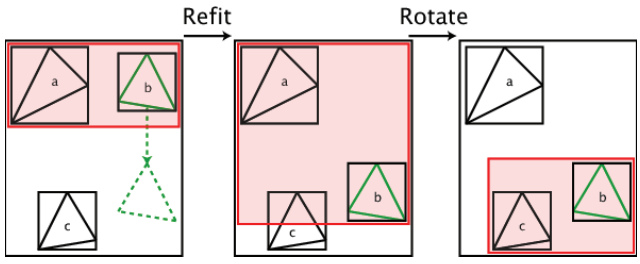


Figure 2: *The effects of refitting and one possible rotation on a simple subtree*

full high quality SAH sweep rebuild on each frame, and performing an approximate rebuild using SAH binning. Per frame SAH sweep builds are impractical due to their lengthy build time, but the resulting tree is of very high quality. An ideal update algorithm would produce these SAH trees instantly every frame, and so we simulate this ideal but non-existent update algorithm by subtracting the rebuild time from the frame time when using a per frame SAH sweep build. This represents the theoretical best case that all algorithms should strive to meet. The binned SAH build is faster, but still too slow; however, parallel binned SAH builds have been used with some success. In practice scalability has been an issue with even very fast implementations achieving 50–75% efficiency [Wald 2007]. Assuming future hardware and algorithms could allow for perfect scalability to the 8 cores we tested, we would like to know whether idealized parallel binned SAH rebuilds would allow for faster frame times than refitting with rotations or just refitting. To simulate this, we took the frame time using the serial binned SAH rebuild and divided the rebuild time component by 8 in order to get a frame time for a perfectly scaling binned rebuild.

Figure 6 shows the time to render a frame over the course of a 100 frame long animation for each of the test scenes. We allowed the animations to loop $2.3\times$ in order to show that rotations are fairly stable even when changing between the end and start key frames which usually are very different. As expected, using tree rotations is never faster than the ideal update, but we can often get fairly close to that ideal and are almost always faster than the parallel binned rebuild. Rotations are, also as expected, almost always better than the refit-only approach.

For the smaller, simpler scenes our rotation and refit algorithm performs very well, tracking closely to the ideal full rebuild performance. In both of these examples the idealized parallel build updates are significantly slower because the amount of deformation is low enough that rotating and refitting are able to keep the SAH cost close to optimal, as evident by the SAH costs in Figure 6. Rotations, and even simple refitting in the case of the Cloth Ball, are able to achieve rates very close to the ideal update frame time.

For scenes with extensive deformation, regardless of size, simple refitting results in very poor quality trees as evident by the orders of magnitude increases to the SAH cost as the animation progresses and the very high time to render a frame. With rotations, the SAH cost is kept much lower than refitting, the frame times never blow up as happens with refitting-only, and in fact the frame times often continue to stay close to the ideal update performance.

The exception is the extremely chaotic BART Museum animation. For this animation, while rotating is much better than refitting-only, it still performs significantly worse than per frame rebuilds. In fact, while not shown, even our single threaded approximate per frame rebuild was able to achieve better frame rates than with rotations. This is due to the initial tree being of very poor quality, as evident by the SAH cost of even the ideal update being as poor as the worst cost that refitting achieves in the Lion and Exploding Dragon and Bunny animations. Given an extremely poor quality tree, tree rotations are able to help but are not able to fix the inherently bad tree and so rebuilding from scratch in this particular case is much better. However, this is rare in practice, as evident that it required a synthetic test to expose this limitation. Building a tree starting from one of the other key frames would have given drastically better results. Another option is to use the rebuild heuristic of [Lauterbach et al. 2006] alongside rotations and force a parallel binned rebuild to occur if the tree quality ever becomes extremely poor. Since in this case performing a rebuild is already a good option, this would not result in a stall and would likely result in overall better frame rates than per frame rebuilds.

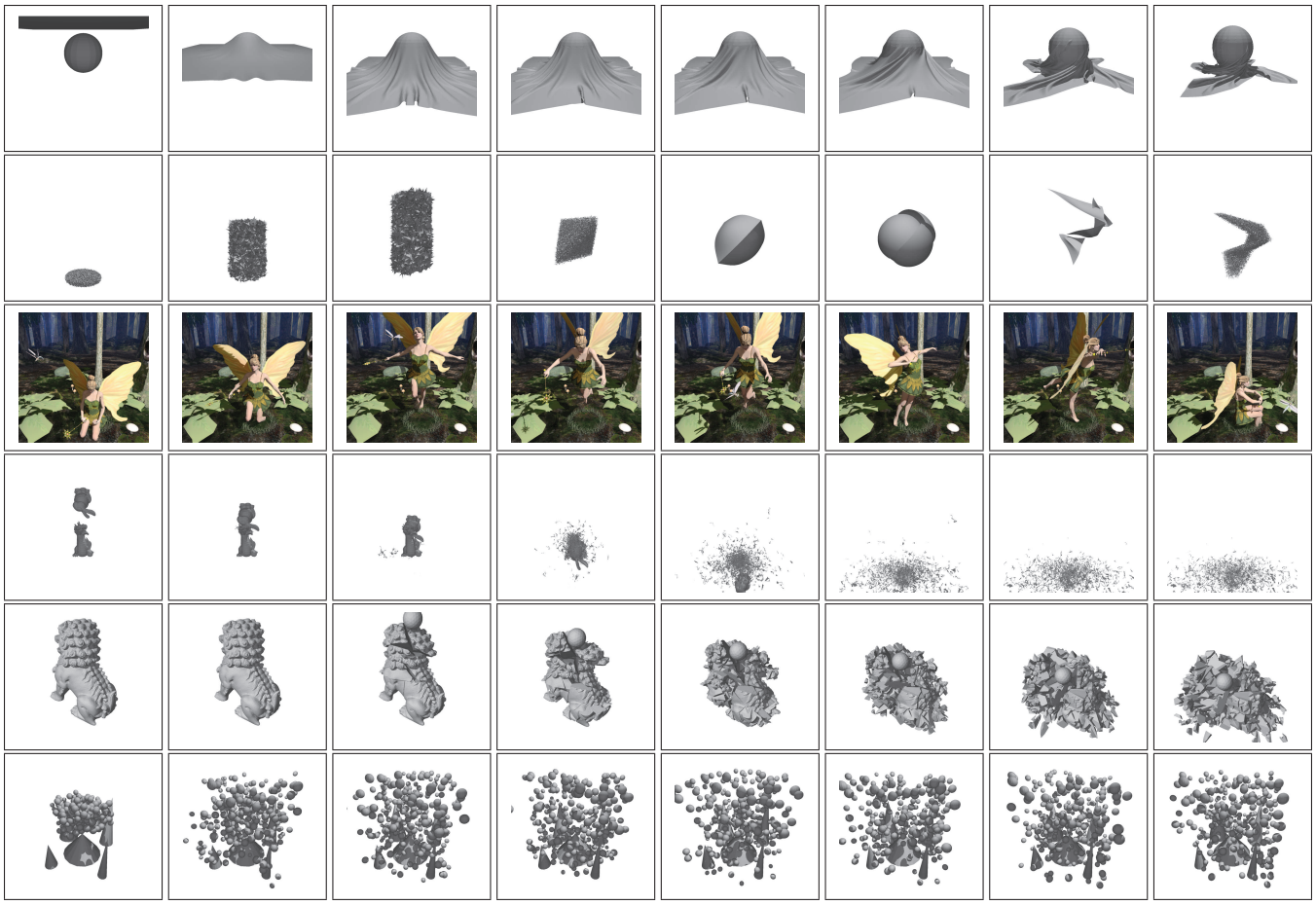


Figure 3: *The 6 animations we used from top to bottom are Cloth Ball (92K tri), BART (66K tri), Fairy Forest (174K tri), Exploding Dragon and Bunny (253K tri), Lion (1.6M tri), and N-body Simulation (146K tri).*

Another anomaly can be seen in the Lion scene. Compared to ideal parallel binned SAH rebuilds, rotations and even simple refitting are significantly faster in the Lion animation due to the large cost of rebuilding that many triangles compared to the relatively quick refit/rotation updates and time to render. For per frame rebuilds to become competitive in large models, the amount of degeneracy introduced by animation must be extremely severe—the lion scene already has an extremely high SAH cost for refitting, and yet even there it just matches the ideal parallel build time—or the amount of rendering work must be very high, such as with non-interactive path tracing, so as to make rebuild time a minor cost.

In order to get an idea of how expensive each update algorithm is to perform, we investigated what the overhead cost is for each technique. Figure 4 shows average time to update each frame for refitting, refitting with rotations, and the ideal parallel approximate rebuild for each of our test scenes. We can take the average of all frames because the times for each algorithm are fairly insensitive to which specific frame it is updating. Adding rotation to refitting only increased the update time by a nominal 1.6–2× and is significantly lower, usually by an order of magnitude, than even an idealized parallel binned rebuild.

Though rotations are significantly more computationally expensive than refitting, it ends up increasing the refitting update cost by less than a factor of two. This is partly due to data sharing between refitting and rotating. Refitting alone must bring every node in to the cache at some point, and when we apply rotations on top of that,

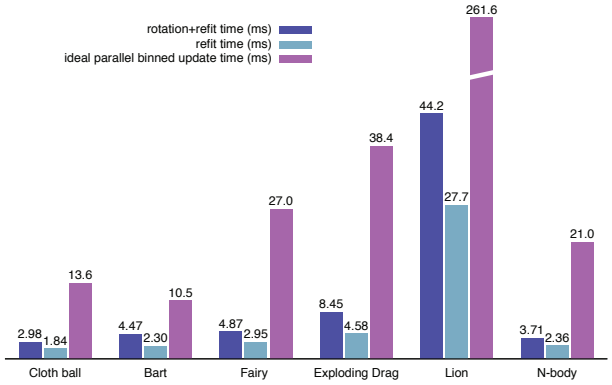


Figure 4: *Average time to update the BVH using refitting, refitting+rotations, and an "ideal parallel binned" rebuild that scales perfectly to all 8 cores.*

we use the same node immediately after it has been refit, taking advantage of temporal locality in the data. We also note that only about one quarter of the total nodes are candidates for rotating. Leaf nodes need not be rotated, cutting out the bottom level of the tree, corresponding to about half of the nodes. In fact, the only nodes that can be rotated are nodes with grandchildren, cutting out the next

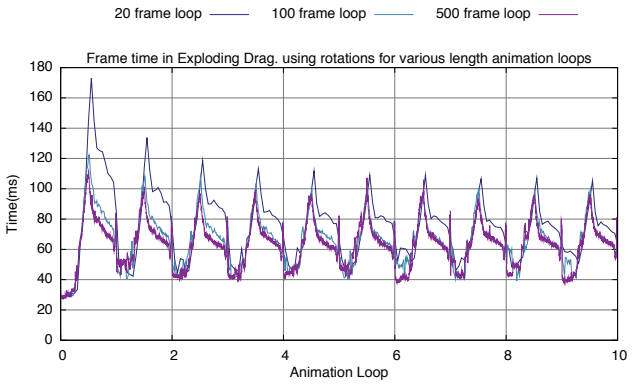


Figure 5: *Frame time for the Exploding Dragon and Bunny when rendered using tree rotations where each animation loop consists of 20, 100, or 500 frames.*

level of the tree as well, and another half of the remaining nodes.

The difference in the tree between one frame and the next can have an effect on the behavior of tree rotations. This is not the case with a rebuild, since it simply constructs a new tree based on whatever position the triangles are currently in. With tree rotations however, since we first refit the nodes from the old tree, the amount of motion that occurred between one frame and the next will determine the quality of a refit tree. The behavior of rotations will differ based on the new quality of the refit tree. To examine this, we ran tests varying the number of frames in one loop of an animation, thus varying the amount of change the geometry undergoes from one frame to the next. Figure 5 shows the frame time for the very chaotic Exploding Dragon and Bunny animation when rendered using tree rotations with 20, 100, and 500 frames per animation loop. Assuming an ideal 60fps for running the animation, the 20 frame loop would take an extremely quick 0.33 seconds, the 100 frame animation would take 1.66 seconds and the 500 frame animation would take a very slow 8.33 seconds. The 20 frame loop has significant tree deterioration between frames since the animation time step is larger between frames, and so is more challenging for our algorithm; however it still performs well compared to the other update methods. The 100 and 500 frame long loops exhibit similar performance, indicating that the tree rotations had enough time to converge within the animation loop. In addition to showing that our algorithm is fairly robust to animation speed, it is also interesting to note that if the animation is allowed to loop, tree quality can continue to improve so that after just a few iterations the 20 frame loop has a tree quality almost as good as the slow 500 frame loop.

4 Discussion

While our algorithm does a good job at preventing the tree from blowing up as can happen with refitting alone, for certain animations that start with an already extremely poor BVH and continue to have very incoherent motion, such as the BART Museum animation, it is not the algorithm of choice. In this case the tree quality is so poor that it is faster to rebuild and render using a new tree than it is to just render using the degenerate tree.

For models with significant deformation that also happen to have low triangle counts, rebuilding per frame using a highly optimized approximate parallel build algorithm can work well; however, for larger models, as commonly found in modern games and scientific visualizations, the per frame rebuilds are too expensive and rotations become significantly better. This is clear in the exploding dragon and lion animations; with the 252K triangle exploding dragon, the parallel build and rotation method both have comparable worst case performance, although the rotation method is significantly faster when the deformation is low. For the 1.6M triangle lion animation, however, parallel rebuilds introduce a significant overhead and are always slower than using rotations. Even for smaller models when the BVH can be built more quickly, rotations win out over rebuilds if the deformation in the animation is not very severe, since they require such little overhead and yet can maintain a high quality tree.

One inherent advantage of our algorithm is its ability to continually improve the quality of the tree on each frame, potentially even beyond that of a fresh build. Since the greedy top-down build is only an estimate of the best build, we can fix up some of the bad estimates by knowing the true SAH costs during the rotation pass. We can see this in our results for the Fairy Forest animation, in which tree rotations produce a higher quality tree than a sweeping rebuild. This will become more pronounced if the animation settles down and motion becomes minimal for some time, or even stops, since rotations are continually applied on each frame.

5 Conclusion

We have presented a fast, lightweight BVH update algorithm that can maintain high quality trees on every frame. For all but one of the 6 animated scenes we tested, our algorithm roughly matches or outperforms, both in average frame time and worst case frame time, the other commonly used techniques for dynamic BVHs. It was even able to match or outperform an idealized parallel approximate build on all but the BART scene, even though no parallel build implementations actually exist that perform that well. In practice, it is safe to assume that our method will compare even more favorably. Since rotations add only a small cost over refitting, all systems that currently rely on refitting would likely always benefit by adding rotations. Only in extremely degenerate animations, such as BART, would we advocate using a full rebuild, or perhaps a combination of rotations with partial rebuilds [Yoon et al. 2007] or a rebuild heuristic [Lauterbach et al. 2006] to perform a full rebuild only when the quality has deteriorated significantly. Likewise, while for many animations rotations should outperform asynchronous rebuilding [Wald et al. 2008], for sufficiently degenerate scenes for which asynchronous rebuilds might be superior, our method could easily be integrated into the asynchronous rebuilds since asynchronous rebuilding already relies on refitting. In that case we could go more frames without losing too much tree quality which would allow for using fewer rebuild cores, or even only use a fraction of a core for rebuilding, so that more resources can be dedicated to rendering.

Our results suggest that tree rotations should be the default update method when rendering deformable animations with a BVH since they have almost negligible cost, work very well for the vast majority of scenes, and from a software engineering perspective can be easily integrated into any preexisting system that already uses node refitting. Furthermore, if the refit algorithm is already parallelized, which is simple to do, then by inserting tree rotations into the refitting operation, tree rotation updates will automatically be made parallel. While this paper targets CPU-based BVH updates, in the context of GPU ray tracing, since refitting is already used as part of an HLBVH build and the refitting is almost an order of magnitude faster than the overall build ([Garanzha et al. 2011] show that the Stanford Dragon takes 1.0ms to refit and 8.1ms to build), using rotations, which in our CPU implementation are roughly as expensive as refitting, should still be an attractive tree update strategy. This is especially true in scenes with reasonable amounts of deformation, such as the Fairy Forest, where the HLBVH+SAH build might take about $4\times$ longer than the refitting with rotations update and also produces a lower quality tree than using rotations.

Acknowledgements

The Fairy Forest scene is courtesy of the Utah 3D Animation Repository, the ClothBall, DragBun, and Lion scenes are from the UNC Dynamic Scene Benchmarks suite, and the BART Museum scene is from the Benchmark for Animated Ray Tracing suite.

References

BIGLER, J., STEPHENS, A., AND PARKER, S. G. 2006. Design for parallel interactive ray tracing systems. In *Symposium on Interactive Ray Tracing*.

GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. 2011. Simpler and faster HLBVH with work queues. In *High Performance Graphics'11*.

GARANZHA, K. 2008. Efficient clustered BVH update algorithm for highly-dynamic models. In *Symposium on Interactive Ray Tracing*, 123 –130.

GOLDSMITH, J., AND SALMON, J. 1987. Automatic creation of object hierarchies for ray tracing. *Computer Graphics and Applications, IEEE 7*, 5 (may), 14 –20.

GÜNTHER, J., FRIEDRICH, H., WALD, I., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum 25*, 3 (Sept.), 517–525. (Proceedings of Eurographics).

IZE, T., WALD, I., AND PARKER, S. G. 2007. 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*, 101–108.

KENSLER, A. 2008. Tree rotations for improving bounding volume hierarchies. In *Symposium on Interactive Ray Tracing*, 73 –76.

KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598, 671–680.

LAUTERBACH, C., YOON, S.-E., MANOCHA, D., AND TUFT, D. 2006. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Symposium on Interactive Ray Tracing*, 39–46.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2, 375–384.

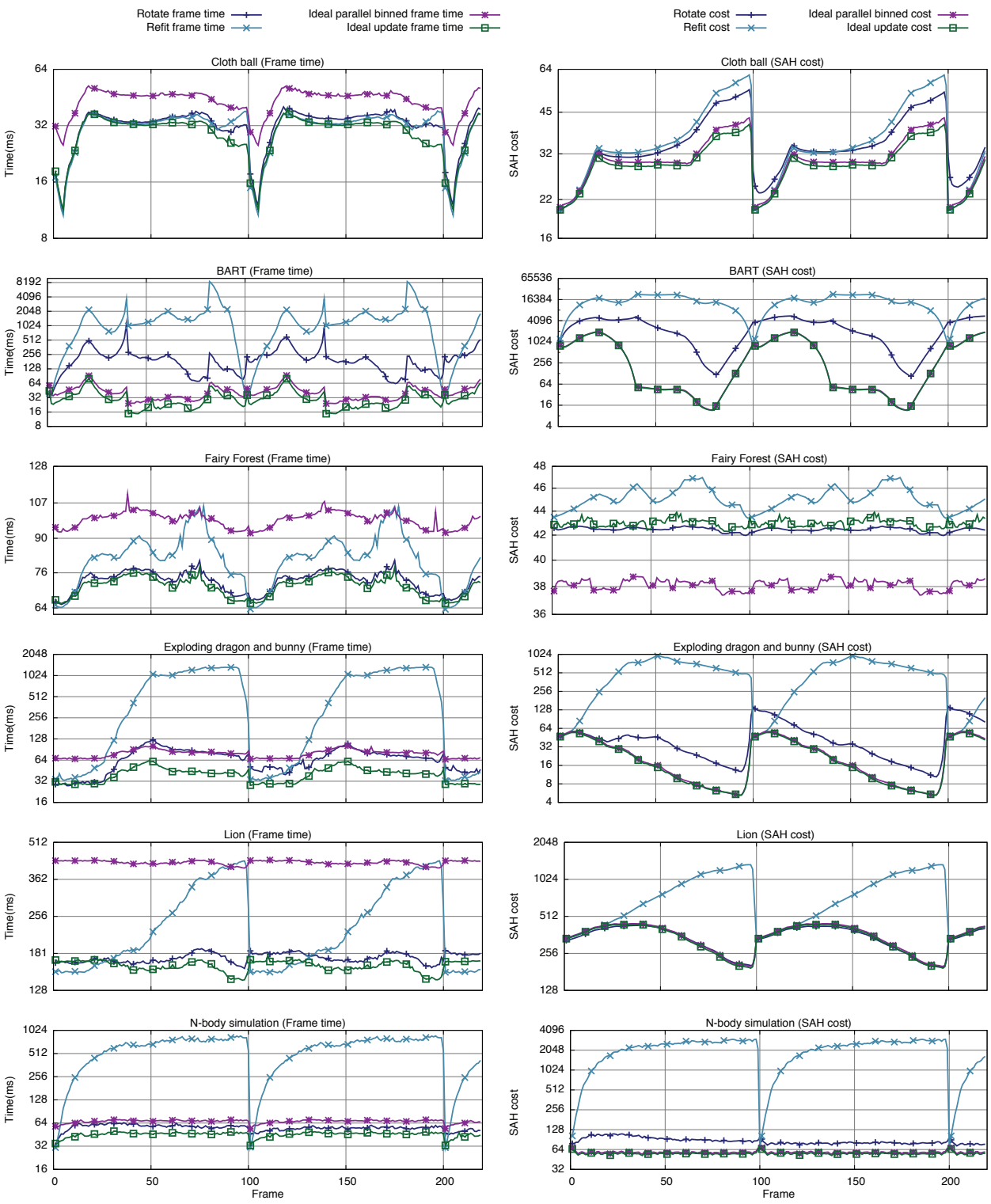
PANTALEONI, J., AND LUEBKE, D. 2010. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *High Performance Graphics'10*, 87–95.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1.

WALD, I., IZE, T., AND PARKER, S. G. 2008. Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics 32*, 1, 3–13.

WALD, I. 2007. On fast construction of SAH based bounding volume hierarchies. In *Symposium on Interactive Ray Tracing*.

YOON, S.-E., CURTIS, S., AND MANOCHA, D. 2007. Ray tracing dynamic scenes using selective restructuring. In *ACM SIGGRAPH 2007 sketches*, ACM, New York, NY, USA, SIGGRAPH '07.



(a) Frame time

(b) SAH cost

Figure 6: *Performance results for the 6 test scenes we used. Each animation was run for 100 frames, then looped 2.3×. The "Ideal parallel binned" algorithm is a binning BVH build that we simulate to unrealistically parallelize perfectly to 8 cores. The "Ideal update" algorithm is a simulation of a sweeping rebuild that happens instantly.*