Fast, Effective BVH Updates for Animated Scenes

Daniel Kopta* University of Utah
Thiago Ize University of Utah
Josef Spjut University of Utah
Erik Brunvand University of Utah
Al Davis University of Utah
Andrew Kensler Pixar

Abstract

Bounding volume hierarchies (BVHs) are a popular acceleration structure choice for animated scenes rendered with ray tracing. This is 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 and a commensurate reduction in the frame rate. The BVH could be rebuilt on every frame, but this could take significant time. We present a method to efficiently extend refitting for animated scenes with tree rotations, a technique previously proposed for off-line improvement of BVH quality for static scenes. Tree rotations are local restructuring operations 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, parallelizes easily, avoids significant degradation in tree quality or the need for rebuilding, and maintains 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 animated scene.

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 between frames, the existing acceleration structure must be either updated or replaced with a new one, the latter of which can be costly. In recent years, bounding volume hierarchies (BVHs) have been a popular subject for research on efficient acceleration structure update algorithms [Wald et al. 2009]. 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.

In this paper, we propose a technique for incrementally updating a dynamically changing BVH to keep the surface area heuristic (SAH) [Goldsmith and Salmon 1987] quality and frame rates high as the geometry moves and deforms. We start with tree rotations, a modification of a technique proposed by Kensler as an offline iterative pre-process to incrementally improve the quality of already high quality static trees [Kensler 2008]. In this case, our context is quite different since our goal is to support animation. We add a single rotation iteration to each frame's refit phase to help maintain tree quality for continuous geometric changes. We find that tree rotations are very effective at rearranging nodes in the BVH for animated scenes where incoherent geometric motion is a common case. In addition to the rotation, we also explore splitting and merging BVH leaf nodes. As primitives move apart, it makes sense to split those nodes to keep the size of the bounding boxes small. Likewise, we should combine nodes as primitives move closer to each other. We also explore the degenerate case of pre-splitting the geometry completely to one primitive per leaf node at build time to accommodate arbitrary scene deformation, which avoids the cost of dynamic splitting and merging. By efficiently folding tree rotations into the per-frame refitting operations, we are able to dramatically improve the quality of the trees generated by a BVH refit with only a small increase in additional processing time. This results in better animation performance than with refitting alone or per frame parallel BVH rebuilds.

1.1 Previous Animation BVH Rebuild Approaches

Full rebuild algorithms replace the BVH with a new one before the change in quality 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. Although this work focuses on BVHs, rebuild strategies also exist for kd-trees [Popov et al. 2006; Hunt et al. 2006].

[Ize et al. 2007] perform the rebuild asynchronously while refitting so that the tree quality never degrades too much and rendering stalls never occur as with the method of Lauterbach et al. However, the asynchronous rebuild requires a dedicated rebuild core, requires significant changes to the ray tracing system, adds an additional one frame lag for 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 produces lower quality trees than those used in a high quality sweep or binned SAH. Furthermore, even these very fast parallel builds are still significantly slower than refitting and do not scale well to many cores. This 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 results in fairly good performance for animations with significant deformation. However it still has the disadvantage that performance can significantly degrade between new trees. For animations with less deformation, inferior performance can result when there are cores dedicated for building the BVH in addition to just refitting.

*email: {dkopta, thiago, sjosef, elb, ald}@cs.utah.edu

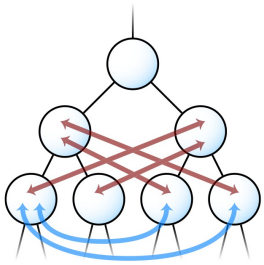


Figure 1: *Potential tree rotations considered*

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 prior to reconstruction.

On a GPU, [Lauterbach et al. 2009] showed how a BVH could be built quickly using the LBVH algorithm. However, tree quality is significantly inferior to that produced by a standard SAH build, especially if the model does 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 and suffers when triangle distributions are nonuniform. 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.

1.2 Tree Rotations for Static BVHs

Tree rotations are local restructuring operations that modify subtrees of a binary tree by swapping direct child and grandchild nodes. A tree rotation lowers one subtree while raising another. They are used in self-balancing binary search trees as a means of rebalancing the tree once it has changed shape. In this application we use them for a slightly different purpose. While we do use them to change the layout of the tree, it is not to achieve balance, it is to achieve lower SAH cost. Lowering SAH cost may actually unbalance the tree, but in a way that is beneficial in terms of overall render time.

Tree rotations for BVHs were first introduced in [Kensler 2008], where rotations are applied as a pre-process on top of an offline build algorithm to improve the quality of the BVH for static scenes. The rotations themselves are slightly modified versions of classical tree rotations, due to the nature of applying them to a BVH tree instead of a binary search tree. The algorithm starts with a BVH built from a full high quality SAH sweep construction [Wald et al. 2007]. It then considers potential improvements to the tree via 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 intermediate steps which temporarily raise the SAH cost. Note that the figure is asymmetric 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, this would result in redundant work.

Since the BVH is built 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.

2 Incremental Updates via Tree Rotations

We recognize that for static scenes, tree rotations can only minutely improve the cost of the BVH tree. However, in an animated scene, as soon as geometry begins to move and nodes are refit, the tree's original configuration is far from optimal. Tree rotations are easily able to find beneficial swaps, rearranging the tree to better fit the new geometry positions. This is illustrated in Figure 2. Tree quality becomes poor after the geometry moves and a simple rotation is able to restore the quality.

On top of refitting and rotations, we also consider allowing leaf nodes to split apart into multiple smaller nodes. During the course of an animation, two triangles that originated close to each other may travel far apart, increasing the volume of the bounding box, and making it potentially beneficial to split the triangles. A node is split by performing a fast approximate binned SAH build on the leaf node [Wald 2007]. If the build determines the node is already fit as a leaf, nothing is done. Otherwise, the leaf becomes an interior node with a small number of descendants. If the leaf has only two primitives, the splitting decision is simpler, and need not go through the SAH build process. If the leaf has only one primitive, it is not a candidate for splitting. After a split has happened, tree rotations take over and separate the triangles into the subtrees where they belong.

As the animation progresses and splits and rotations are applied, geometry will find itself grouped into subtrees with new partners. Rotations will tend to place primitives that are spatially next to each other into the same subtree. We therefore allow two sibling leaf nodes to merge into one larger leaf node if it results in a lower SAH cost. As the geometry moves further, they are free to split apart again and move about the tree.

Our update algorithm is easily applied on top of the traditional refitting operation. 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. During this traversal, our update steps can be added after the refit for each node. If the node is a leaf, we check if it is a candidate for splitting. If the node is a parent of two leaves, we consider merging them into one. For each node with at least two levels below it, we attempt to find a beneficial rotation. After rotation we update the bounding volume of the affected child node with another refit operation that tightly bounds its new children. This is essentially an incremental partial sort of the scene geometry per frame, as opposed to a full sort (rebuild).

Figure 2 illustrates the process of refitting and rotation. We show both the spatial representation of the bounding volumes and the structural representation of the tree. The node that we are considering for rotation is the outer blue bounding node that contains all three triangles in the example. Before moving, the leaf node containing

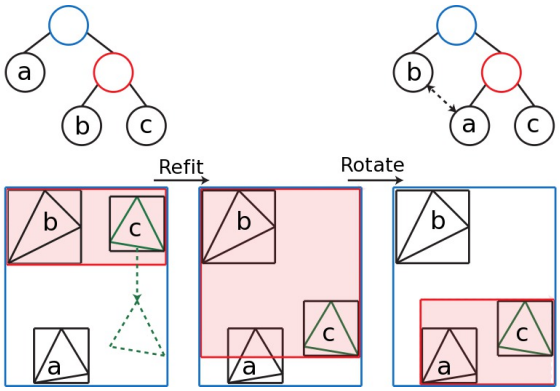


Figure 2: *The effects of refitting and one possible rotation on a simple subtree*

the green triangle (c) was originally grouped in a subtree with leaf node (b) as its sibling. After (c) moves, the red parent node is refit. 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 either of the two nodes contained within that subtree, and 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 (b) produces a tree with a lower SAH cost because the new red parent node contains less empty space. This process is done from the bottom up for every node in the tree that has grandchildren.

Figure 2 also illustrates the potential for splitting and merging nodes as geometry changes. In this case consider the red shaded node as the leaf node instead of the black node. As triangle (c) moves and expands the bounding volume, it would make sense to split the node into separate nodes containing triangles (b) and (c). In a merging stage it could then make sense to combine nodes (a) and (c) into a new red leaf node. Tree rotations seem to be ideal for modifying the tree in a way that makes helpful merges more apparent.

Some animations rearrange the geometry so drastically, that most of the nodes will need to be split. With unpredictable motion, nodes that are merged together may end up being re-split a few frames later, resulting in wasted work. We therefore consider another technique of fully pre-splitting nodes down to a single primitive per leaf at build time, and force them to remain that way by turning off splitting and merging. This eliminates the work of checking for beneficial merges and performing an approximate SAH build on each leaf node, with the side effect of higher initial SAH tree cost.

Since tree rotations can be implemented with a post-order traversal and work on the same data visited during refitting, we can remove most of the memory access costs incurred by tree updates by performing all of the refitting, rotations, splitting, and merging 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 our proposed updates will also be parallelized simply by adding them in to the refit.

3 Results

We tested our algorithm in the Manta interactive ray tracer [Bigler et al. 2006]. This code is also now included in the official Manta source code repository. Starting with the existing recursive refitting code in Manta, we extended this to also maintain cost evaluations, have the ability to split and merge BVH leaf nodes, 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 eight-core Intel Xeon X5550. All scenes were rendered at 1024×1024 pixels with shadows for a single point light source. All refitting, rotation, split, merge updates, and renderings used eight parallel threads. For BVH traversal, we use 8×8 ray packets and the interval arithmetic culling scheme described in [Wald et al. 2007].

Table 1: *Average frame time in ms (update + render) for baseline techniques and our proposed update techniques. Overall combined fastest time (update+render) is in bold. The final column is a percentage improvement from the pre-split rotate scheme to refit-only.*

Scene	Refit	Split-Merge refit	Split-Merge Rotate	Pre-split Rotate	Comp. to Refit
Clothball	**4.9 + 14.4**	6.4 + 13.4	7.8 + 12.0	6.7 + 13.2	-3%
BART	4.3 + 1948.4	6.9 + 762.2	**10.0 + 115.0**	8.7 + 119.8	+1420%
FairyForest	7.4 + 53.6	11.7 + 52.5	**14.3 + 45.1**	**11.4 + 48.9**	+1.2%
DragBun	12.2 + 388.3	19.6 + 201.8	**24.2 + 21.2**	22.3 + 23.8	+769%
Lion	65.0 + 151.9	98.4 + 65.8	120.4 + 9.6	**101.3 + 23.8**	+73%
N-body Sim	6.5 + 354.3	8.4 + 285.3	**10.1 + 20.8**	9.6 + 22.3	+1031%

Our example animations, seen in Figure 3, fall into two broad categories: smaller scenes with simple deformations (Clothball (92k triangles) and Fairy Forest (174k)), and scenes with a variety of sizes, but with extensive deformations (Exploding Dragon and Bunny (253k), Lion (1.6M), N-Body Simulation (146k), and BART Museum (66k)).

3.1 Tree Rotations with Node Splitting and Merging

Our initial tests measure performance on each animation using refitting alone (Refit), refitting with splitting and merging (Split/Merge Refit), adding rotations to splitting and merging (Split/Merge Rotate), and finally the degenerate case of pre-splitting down to a single primitive per leaf at build time plus rotations (Pre-split Rotate). We find that the pre-split tree allows primitives to move about the tree more freely and presents more rotation opportunities for the moving geometry than the non-split tree. The result is a similar SAH cost in the two cases. When combined with the overhead savings of turning off splitting and merging, the performance after rotations on a pre-split tree and on a tree that includes splitting and merging is similar.

Table 1 summarizes the results. We found that for the very well behaved Clothball scene, refitting alone did fine and adding rotations increased the overall frame rate by an insignificant amount. Adding splitting and merging of leaf nodes on top of refitting had a beneficial effect on most of the scenes. We can see that the overhead of the update increases, but in most cases also results in a substantial decrease in render time.

For all the scenes except Clothball, adding rotations had a positive, often substantial, impact on frame rates. Even on the Clothball scene, the performance difference is negligible, and adding rotations is still a safe option. While splitting and merging nodes in addition to rotations does improve performance in some cases, we found that the benefit is always minor. We can see that splitting and merging does reduce render time in all cases due to a slightly higher quality tree, but at the cost of greater update times. The tradeoff between the two techniques is roughly equivalent.

Detailed results can be seen in Figure 6 for the Dragon-Bunny scene. The other scenes had similar overall characteristics, but with different absolute values. The number of nodes in the BVH does go

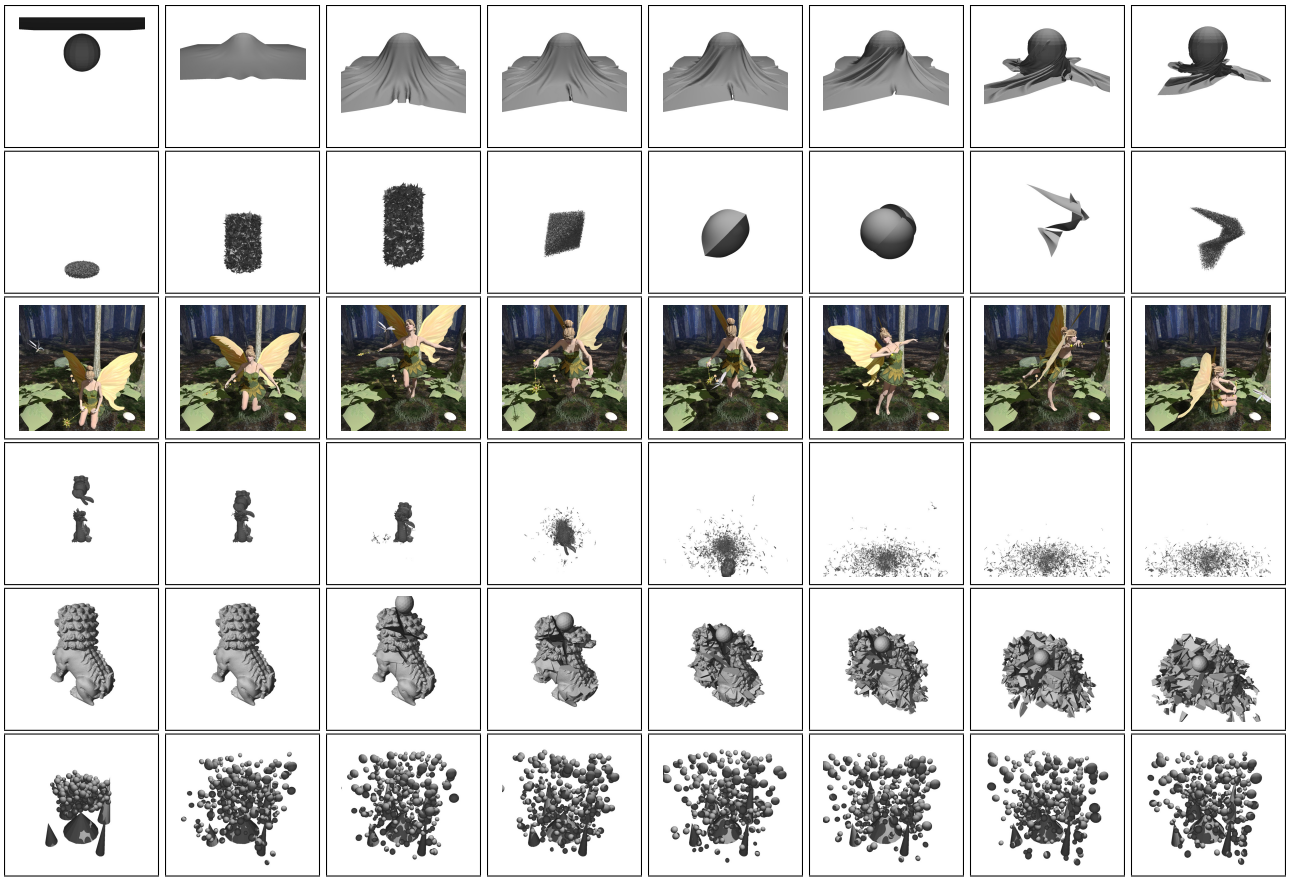


Figure 3: *The 6 animations we used from top to bottom are Cloth Ball (92K tri), BART (66K tri), Fairy Forest (174K tri), Exploding Dragon and Bunny (253K tri), Lion (1.6M tri), and N-body Simulation (146K tri).*

down when splitting and merging is used because multiple primitives are contained in each leaf node. However, because of the additional overhead of splitting and merging, the update time for each frame is higher. Interestingly, the SAH cost is almost the same in both cases. This is partly due to the fact that there are more rotations performed per frame with the pre-split technique. The difference in render time is also partly due to the greater number of nodes in the pre-split tree that must be potentially traversed before hitting a leaf. We do not include the non-rotating methods in Figure 6 simply because the performance is so poor and the SAH cost is so high, it detracts from the comparison of the two interesting contenders.

All six scenes had similar behavior: the update cost for splitting, merging, and rotating were higher than for just rotating, but the total frame time was very similar and there was no consistent winner. Because of this, and because the rotations alone are much simpler to implement, we chose to focus on the pre-split (one primitive per leaf) BVH for our more detailed comparisons against other BVH update techniques in the next section. We note, however, that if BVH size is a primary concern, the merging and splitting option does reduce the number of nodes substantially and performs just as well.

3.2 Tree Rotations on a Pre-Split BVH

Based on the results in the previous section, we have performance data for all six scenes using refitting only, and refitting with rotations on a pre-split BVH. For comparing to an ideal scenario, we measured the performance on each animation against two baseline techniques: performing a 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. 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 on our eight core test case, 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 eight in order to get a frame time for a perfectly scaling binned rebuild.

Figure 7 shows the time to render a frame over the course of a 100 frame animation for each of the test scenes. We allowed the animations to loop $2.2\times$, 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 (still unrealistic) parallel binned rebuild. Rotations are 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 7. Rotations, and even simple refitting in the case of the Clothball, 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 required 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 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 idealized 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 poor quality of the initial tree. The SAH cost of even the ideal update for BART is worse than the lowest 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. Fortunately, this is rare in practice and we had to employ a synthetic test to expose the 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. 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.

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 with many triangles compared to the relatively quick refit/rotation updates and rendering time. For per frame rebuilds to become competitive in large models, the amount of degeneracy introduced by animation must be extremely severe. The lion scene has an extremely high SAH cost for refitting, and yet it just matches the ideal parallel build time. Alternatively, the amount of rendering work must be very high, such as with non-interactive path tracing in order to make the rebuild time a minor cost.

To gauge the cost of each update algorithm 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 the specific frame. 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.

Although 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 into the cache at some point, and rotations reuse the same node immediately after it has been refit. This temporal locality is cache friendly. We also note that only about one quarter of the total nodes are candidates

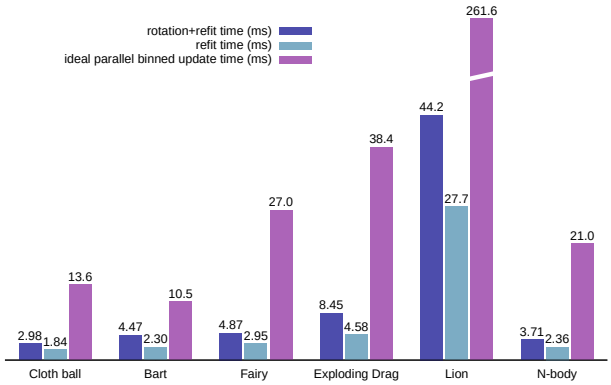


Figure 4: *Average time to update the BVH using refitting, refitting+rotations, and an "ideal parallel binned" rebuild that scales perfectly to all 8 cores.*

for rotating, since a node must have grandchildren in order to apply a rotation. This cuts out the bottom two levels of the tree.

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, which fully 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 between frames. 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 due to the large animation time step between frames. Though more challenging for our algorithm, it still performs quite well. The 100 and 500 frame long loops exhibit similar performance, indicating that tree rotations had enough time to converge within the animation loop. In addition to showing that our algorithm is fairly robust to animation speed, tree quality can continue to improve if the animation is allowed to loop. For example, after just a few iterations the 20 frame loop has a tree quality almost as good as the 500 frame loop.

3.3 Tree Rotations on the GPU

We also investigate the application of our algorithm on GPUs, as they are commonly considered an interesting platform for ray tracing. The massive parallelism and SIMD execution of a GPU presents a challenge because of the way that work is assigned to threads. The naive approach of assigning each thread an equal-sized portion of the tree and performing a post-order traversal presents difficulties because individual portions of the tree are very small due to the large number of threads requiring work assignments. This leaves a large portion of the top of the tree unprocessed, which can not be as easily parallelized. Furthermore, individual threads in a SIMD warp tend to diverge in control flow due to the variance of each assignment, causing a large portion of potential threads to be inactive.

To alleviate this, we perform a bottom-up traversal of the tree. If the nodes of the tree are sorted by height, simply traversing the

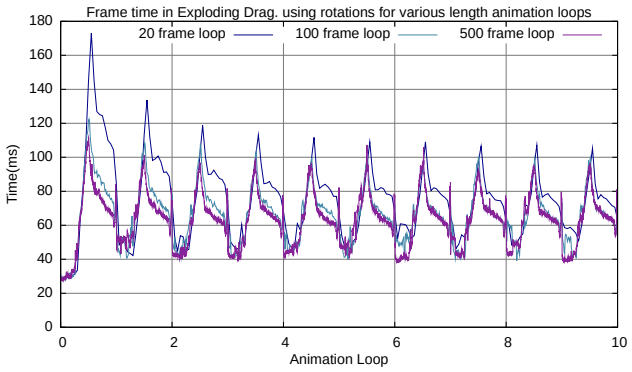


Figure 5: *Frame time for the Exploding Dragon and Bunny when rendered using tree rotations where each animation loop consists of 20, 100, or 500 frames.*

nodes from first to last will guarantee that children are visited before their parent, which is necessary for refitting. However, there is no requirement that nodes at the same height in the tree be visited before any other node at that height. They can therefore be updated in parallel. To take advantage of this, we store the height of each node (leaves being at height 0), and order the references to the nodes by height. We then execute a GPU kernel with as many threads as nodes at that height, and repeat for each height of the tree from bottom to top. This is similar to the refitting procedure used in [Garanzha et al. 2011], although we traverse nodes in order of height, instead of in order of level (depth). This way all leaf nodes are processed in parallel and will never be processed at the same time as an interior node. Control flow divergence will be low among the threads, leading to high SIMD utilization.

Applying tree rotations on top of this pass over the tree has the unfortunate side-effect that the nodes are no longer stored in height order, since we must update the affected nodes' heights after a rotation is done. To solve this, we use the parallel radix sort described in [Merrill and Grimshaw 2011] on the node references, using their heights as keys, to restore their correct order. We compare this technique to the fastest known BVH builder on a GPU, HLBVH [Garanzha et al. 2011]. The only animated benchmark scene in that work is the Fairy Forest, so we compare to their results for that scene using the same device, an NVIDIA GTX480 GPU. HLBVH builds the Fairy Forest in 4.8ms, with a tree quality slightly worse than that of a full SAH build (about 94%). Our pre-split tree rotation update takes 3.7ms, with the same SAH quality as our CPU implementation. Though Garanzha et al. do not give their actual SAH costs, it is likely that the quality of the tree proposed here is higher than that produced by their HLBVH build. Therefore, tree rotations are likely to be an attractive BVH update strategy on GPUs.

4 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 six 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. Since rotations add only a small cost over refitting, all systems that currently rely on refitting would likely 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, for many animations rotations should outperform asynchronous rebuilding [Wald et al. 2008] for sufficiently degenerate scenes. While asynchronous rebuilds might be superior in some cases, our method could easily be integrated into asynchronous rebuilds that already rely on refitting for per frame updates. In that case we could allow the animation to proceed for more frames without losing too much tree quality, which would in turn allow for more CPU time to be used for rendering instead of asynchronous rebuilding.

For models with significant deformation and low triangle counts, rebuilding per frame using a highly optimized approximate parallel build algorithm can work well. For larger models, commonly found in modern games and scientific visualizations, the per frame rebuilds are too expensive and rotations become a significantly better method. 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, parallel rebuilds introduce a significant overhead and are always slower than using rotations. For smaller models when the BVH can be built more quickly, rotations are an improvement over rebuilds if the deformations in the animation are not severe, because high tree quality is maintained with small overhead.

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 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, where 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 or stops for some time, since tree rotations are continually applied on each frame.

While this paper primarily targets CPU-based BVH updates for which previous update strategies are still significantly inefficient, in the context of GPU ray tracing, our unoptimized tree rotation implementation is already competitive with the state-of-the-art HLBVH update of Garanzha et al. [Garanzha et al. 2011]. Though we leave this as future work, we fully expect that our GPU implementation could be still further improved through low-level optimizations similar to those employed by the HLBVH.

Our results suggest that on a CPU and possibly even on a GPU, tree rotations should be the default update method when rendering deformable animations with a BVH since they have low 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.

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. NVIDIA generously donated the GTX480 hardware. This research was supported in part by NSF grant CNS10174757.

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.

HUNT, W., MARK, W. R., AND STOLL, G. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Interactive Ray Tracing IRT06*.

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.

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.

MERRILL, D., AND GRIMSHAW, A. 2011. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters 21*, 02, 245–272.

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.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Experiences with streaming construction of sah kd-trees. In *Interactive Ray Tracing IRT06*.

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., MARK, W. R., GNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2009. State of the art in ray tracing animated scenes. *Computer Graphics Forum 28*, 6, 1691–1722.

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.

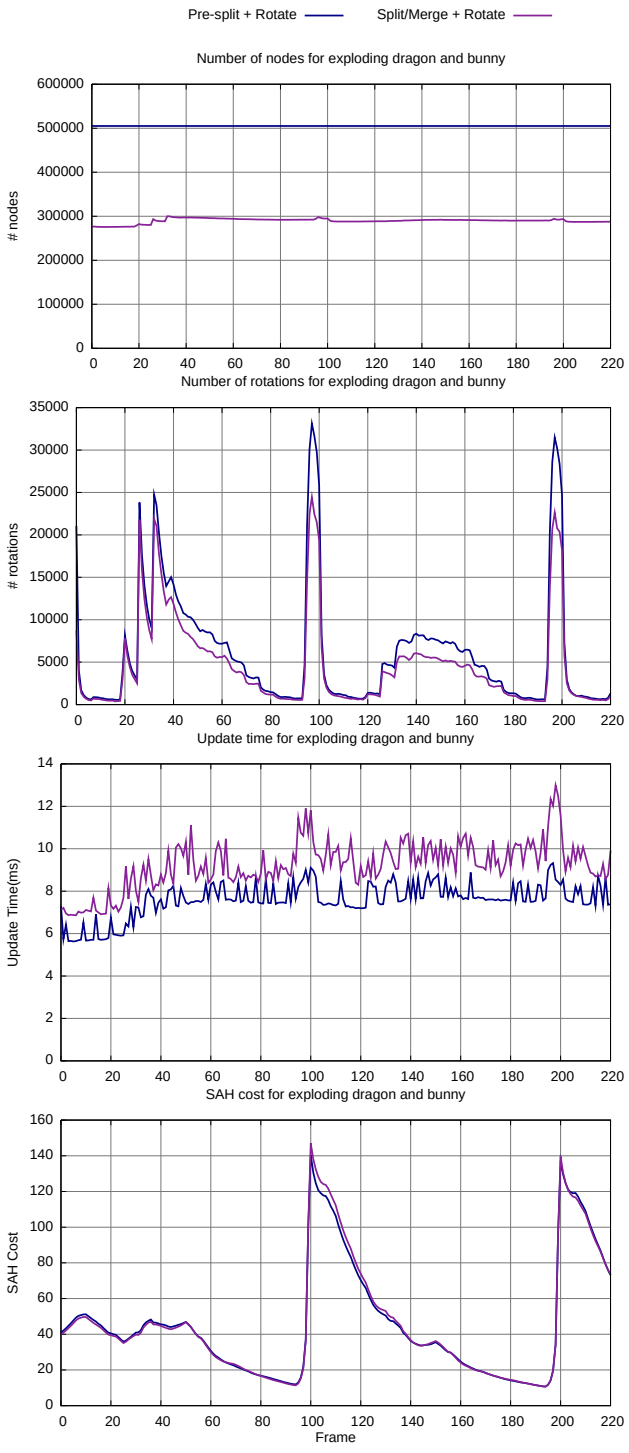
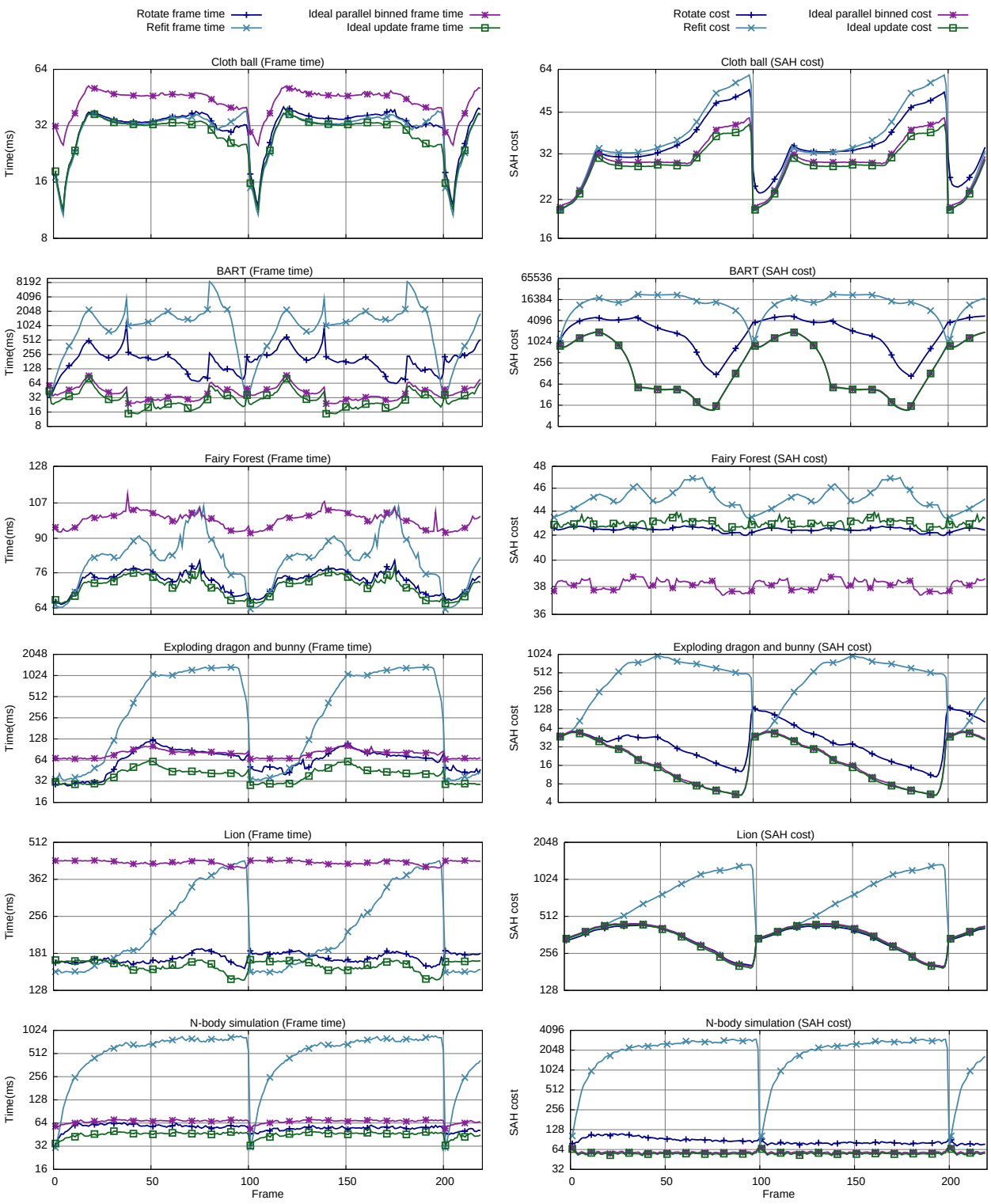


Figure 6: *Performance statistics for splitting/merging + rotating, and pre-splitting + rotating on the Exploding Dragon and Bunny scene. The animation is run for 100 frames and looped 2.2×.*



(a) Frame time (b) SAH cost

Figure 7: *Performance results for the 6 test scenes we used. Each animation was run for 100 frames, then looped 2.2×. 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.*