SCI INSTITUTE
TECHNICAL REPORT

Asynchronous BVH Construction for Ray Tracing Dynamic Scenes

Thiago Ize, Ingo Wald and Steven G. Parker

UUSCI-2006-034

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

November 13, 2006

Abstract:

Recent developments have produced several techniques for interactive ray tracing of dynamic scenes. In particular, bounding volume hierarchies (BVHs) are efficient acceleration structures that handle complex triangle distributions and can accommodate deformable scenes by updating (refitting) the bounding primitive without restructuring the entire tree. Unfortunately, updating only the bounding primitive can result in a degradation of the quality of the BVH, and in some scenes will result in a dramatic deterioration of rendering performance. The typical method to avoid this degradation is to rebuild the BVH when a heuristic determines the tree is no longer efficient, but this rebuild results in a disruption of interactive system response.

We present a method that removes this gradual decline in performance while enabling consistently fast BVH performance. We accomplish this by asynchronously rebuilding the BVH concurrently with rendering and animation, allowing the BVH to be restructured within a handful of frames.

THE UNIVERSITY OF UTAH

Asynchronous BVH Construction for Ray Tracing Dynamic Scenes

Thiago Ize Ingo Wald Steven G. Parker
SCI Institute, University of Utah
{thiago,wald,sparker}@sci.utah.edu

Abstract

Recent developments have produced several techniques for interactive ray tracing of dynamic scenes. In particular, bounding volume hierarchies (BVHs) are efficient acceleration structures that handle complex triangle distributions and can accommodate deformable scenes by updating (refitting) the bounding primitive without restructuring the entire tree. Unfortunately, updating only the bounding primitive can result in a degradation of the quality of the BVH, and in some scenes will result in a dramatic deterioration of rendering performance. The typical method to avoid this degradation is to rebuild the BVH when a heuristic determines the tree is no longer efficient, but this rebuild results in a disruption of interactive system response.

We present a method that removes this gradual decline in performance while enabling consistently fast BVH performance. We accomplish this by asynchronously rebuilding the BVH concurrently with rendering and animation, allowing the BVH to be restructured within a handful of frames.

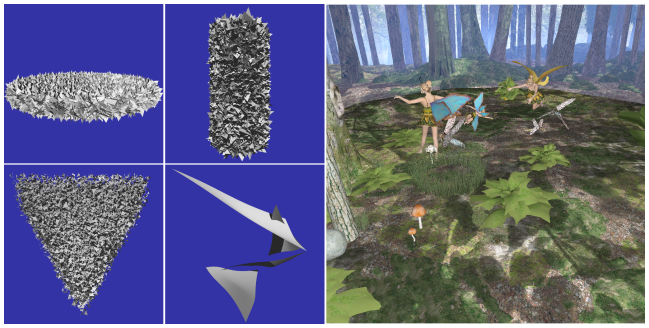

Figure 1: Some example screenshots from our system. a-d) "BART museum" objects of 66K triangles each; even though the scene deforms significantly over time, our approach allows for rendering this scene without degradation over time, and without the freezes introduced by on-demand BVH reconstruction. e) The "Fairy Forest 2" scene (394K triangles total); on an 8-core 2.0 GHz Opteron PC, this scene can be rendered with 19-23 frames per second (1024×1024 pixels, simple shading) and up to 2 times faster than with refitting alone.

1 Introduction

In the last decade, the graphics community has benefited from tremendous improvements in the performance and capabilities of PC based graphics cards, with GPUs now offering hundreds of GigaFlops and increasingly programmability. This demand for faster and more programmable GPUs is driven mainly by the demanding needs of video games for faster and more realistic graphics.

Along with the tremendous improvements in GPUs, CPUs are also becoming much faster, especially with the current trend of increasing the number of cores per chip. For instance, a standard 3 GHz dual-core Opteron today has roughly 24 GFlops, a PlayStation 3's CELL processor has 180 GFlops, and Intel claims to have an 80 core processor prototype capable of TeraFLOP performance [Intel 2006].

The quest for increased quality, combined with increases in available compute power has led to improvements of rasterization-based GPUs. This quest has also reignited an interest in ray tracing for many applications. Ray tracing can fulfill the growing quality demands, but the main limitation is that it is not efficient enough for use in dynamic applications such as games. As long as compute power continues to rise, ray tracing will eventually become real-time.

With this in mind, many researchers have recently focused on realizing real-time ray tracing, and, more recently, on ray tracing dynamic scenes. Today, real-time ray tracing with dynamic scenes can be realized via either kd-trees, grids, or bounding volume hierarchies (BVHs), but there are tradeoffs associated with each of these data structures. Kd-trees seem to offer the highest ray tracing performance, but are most costly to build [Wald and Havran 2006]; grids are efficient to build [Wald et al. 2006b], but rely on a high degree of coherence which may not exist for complex scenes and/or secondary rays. BVHs offer a compromise between performance and the ability to handle complex scenes and secondary rays, but are currently limited for many types of dynamic scenes. In particular, BVH-based interactive ray tracing systems are currently restricted to scenes that deform over time, and will deteriorate in performance for unstructured motion or severe deformations.

In this paper, we propose a new approach for handling dynamic scenes in a BVH-based ray tracer that is designed for highly parallel architectures, and that handle scenes with large deformations over time. In particular, we present an architecture where a subset of CPUs continuously and *asynchronously* rebuild new BVHs (potentially over the course of multiple frames) while the remaining CPUs concurrently update and traverse the existing BVH. This reduces BVH deterioration over time, while avoiding disruptive pauses at BVH rebuild times.

2 Background

Real-Time Ray Tracing and Dynamic Scenes As early as the 1990s, researchers achieved interactive ray tracing performance with the use of large supercomputers [Parker et al. 1999; Muuss 1995; Green and Paddon 1990]. With the growing capabilities of commodity architectures, researchers then turned their attention to ray tracing on PCs, and PC clusters [Wald et al. 2001]. In particular, since Reshetov's "Multilevel Ray Traversal" [2005], PC based ray tracers are—at least for very simple shading—able to achieve fully interactive frame rates for non-trivial scenes on multi-core desktop PCs. In parallel to Reshetovs MLRT approach, Woop et al. [2005] demonstrated that real-time ray tracing can also be achieved by building special purpose hardware.

Apart from low level optimizations, fast ray tracing depends on using an efficient spatial acceleration structures, such as a BVH, kd-tree, octree, or grid. While there is been a long-running debate on which of these is best, by 2005 virtually all fast ray tracers were built on kd-trees. Unfortunately, kd-trees are costly to build

and cannot easily be incrementally updated, and as such present an obstacle to handling dynamic scenes. Thus, the debate is once again open, with researchers actively exploring better ways to support dynamic scenes with other acceleration structures. For kd-trees, Günther et al. [2006] proposed a "motion decomposition" approach to updating the kd-tree, but this only works if the animation sequence is known in advance. Stoll et al. [2006] proposed a lazy build mechanism that is aided by scenegraph information, but no real-time data is yet available. For rendering general dynamic scenes with a kd-tree, Popov et al. [2006] and Hunt et al. [2006] have investigated fast approximate methods to rebuild kd-trees from scratch; these, however, are still rather slow, and seem to parallelize poorly ([Popov et al. 2006] reports no speedup for small models and only a 2.4× speedup on 4 CPUs for a 10M triangle model).

As an alternative to kd-trees, Wald et al. [Wald et al. 2006b] have proposed a grid-based approach to ray tracing dynamic scenes, which can handle arbitrarily dynamic scenes by rebuilding the grid every frame. Ize et al. [2006] have shown that the grid build can be parallelized quite effectively, with interactive rebuild rates even for complex scenes. Unfortunately, this grid-based approach relies on highly coherent ray packets, and extensions for highly complex scenes and/or secondary rays are not obvious.

BVH-based Dynamic Scene Ray Tracing In parallel to the grid and kd-tree based approaches, several groups have investigated the use of bounding volume hierarchies for ray tracing dynamic scenes. Instead of subdividing space into "voxels" of triangle references, BVHs build an *object hierarchy* tree, and in each tree node store a bounding volume for that subtree's geometry. In [2006a], Wald et al. proposed a traversal algorithm for BVHs that achieved performance similar to Reshetov's MLRT system. Concurrently, a similar approach was developed by Lauterbach et al. [Lauterbach et al. 2006]. Other BVH inspired approaches have recently been proposed by Havran et al. [2006], Woop et al. [2006], and Wächter et al. [2006].

All of these BVH-based approaches make use of a BVH's ability to simply "refit" an existing BVH instead of rebuilding it. A BVH is defined through two parts: the tree topology, and each tree node's bounding volume. Once the objects move, instead of rebuilding the complete BVH from scratch, one can also leave the topology intact and only refit the BVH by recomputing all its nodes' bounding volumes. While this refitted BVH may have a different tree structure than one built from scratch, it will nonetheless always be *correct*. Refitting a BVH is extremely fast, and often less costly than the associated animation updates.

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

As pointed out by Lauterbach et al. [2006], deforming a BVH usually works for at least some number of frames, and instead of rebuilding a BVH every frame, one could rebuild only every few frames, with the frames in-between handled by BVH deformations.

In order to do these rebuilds when they are most effective, Lauterbach et al. [2006] have proposed a "rebuild heuristic" that detects BVH degradation, and rebuilds the BVH if and only if the quality degradation has reached a given threshold. This allows for striking a balance between total rebuild cost and render cost, and can yield a significantly reduced average frame time in an animation.

Unfortunately, a lower *average* frame time is not always helpful in an interactive setting. In an offline animation the infrequent rebuilds can be amortized over all frames of the animation, yielding a low average time per frame; in an interactive setting, however, amortization does not apply, and system responsiveness is disrupted while a rebuild is performed, which hurts the user's ability to interact with the environment. For interactive applications, removing large variations in frame rate is often more important than having a moderately faster average frame time [Watson et al. 1998].

3 Asynchronous Dynamic BVHs

As discussed in the previous section, BVHs hold promise for ray tracing dynamic scenes, but the refits eventually lead to degraded performance, and rebuilds result in disturbing pauses. These problems will likely become exacerbated by future ray tracing systems running on architectures with a large number of cores, since the rebuilding does not scale as well as the rendering, thus further worsening the effect of the disruptions.

In this paper, we propose combining fast and *asynchronous* single-threaded BVH builds with parallel and scalable refitting and rendering. We continuously and *asynchronously* build new BVHs as fast as possible in a dedicated thread while all other threads are kept busy with rendering. Even with fast build algorithms, a BVH will usually take more than one frame to build; but since building is asynchronous, rather than wait for the build to finish, the rendering threads can refit the previous BVH and continue rendering, which can easily be done in parallel. As soon as a new BVH is available, the rendering threads switch to the new BVH, and rebuilding starts anew. Using this approach, BVH deterioration is avoided since no BVH is deformed for more than a couple of frames; and because no dependencies between building and rendering are introduced, the rendering threads are never stalled, producing good scalability and avoiding disruptions altogether (see Figure 2).

3.1 Asynchronous Build

To allow multiple threads to work asynchronously on the same data, we must double buffer the shared data, which consists of the vertex positions (which are updated each frame by the renderer) and, of course, the BVH nodes. All other data, like triangle connectivity, triangle acceleration structures, vertex normals, texture coordinates, etc are not touched by the builder, and so are not replicated. This results in roughly 50 bytes extra storage per triangle, which for most scenes has a minor impact.

Whenever a new BVH is finished, the rebuilder passes it to the renderers, and grabs a new set of vertices to work on. This naturally occurs between when the render threads finish their current frame and before they start refitting the BVH for the next frame. While we could wait for the new vertex positions to be calculated before exchanging the data, this would require an expensive copy of those values to the rebuilder, which is especially problematic since the render threads must be blocked waiting for the copy to finish before they can use the new data. This critical section hurts the system's scalability. Instead, before the new vertex positions are calculated, the rebuilder blocks the other render threads, does a couple quick pointer swaps of its vertex and BVH buffer with the renderers, and releases the renderers so that they can calculate the new vertex positions and render the next frame with the new BVH. While the

Figure 2: Given a highly parallel architecture with N parallel threads, we spend N-1 cores on parallel rendering and BVH refitting. The N'th thread works asynchronously and builds new BVHs as fast as possible, potentially over multiple frames (2 in this example); in the meantime, the render threads rely on deforming the most recently finished BVH. BVHs are never deformed for more than a couple of frames, and both scalability bottlenecks and pauses are avoided altogether.

pointer swapping requires that the BVH builder work on already-outdated vertices, since it will take several frames before the new BVH is available, anyway, that BVH will be outdated by the time it is finished, so building the BVH with vertex positions that are outdated by one frame is equivalent to the build taking one frame longer to complete—which is a minor cost for ensuring good scalability. In addition, while the swapping could be performed during the actual rendering, this would require extra synchronization during rendering, which would probably hurt performance more than is made up by having the newer BVH as soon as it is available.

3.2 Parallel Update and BVH Refitting

With the poorly parallelizable BVH build moved to its own asynchronous thread, the rendering stage itself can be kept highly parallel. In particular, the operations to be performed per frame are updating the vertices, refitting the most recently built BVH, updating the triangle acceleration data, and ray tracing. On a machine with N cores, we reserve one thread for the BVH build, and dedicate the remaining $N-1$ threads for parallel updating and rendering.

Vertex Generation The first task is to get the new frame's vertex positions, which are required by the other update steps. Vertices are usually generated by the application using, for example, a vertex shader or linear interpolation. Since for non-trivial scenes even generating the vertices can be quite costly compared to refitting a BVH or rendering a frame, ensuring good system scalability requires either parallelizing the vertex generation, or having the application generate the vertices asynchronously to rendering. In our current framework, we compute vertices by linearly interpolating between fixed timesteps, which we do in parallel on $N-1$ threads.

Parallel BVH Refit Once the new vertex positions are known we can start refitting the most recently finished BVH's bounding volumes; this again has to be done in parallel. One way of doing so is a static work assignment, in which each thread works on one of the top $N-1$ independent subtrees of the BVH. While for two threads Lauterbach et al. [2006] reported good results for that approach, we found that for significantly more threads, and for complex models with uneven geometry distributions, this did not load balance well.

Therefore, we use a three-way dynamic load-balancing scheme for the update. In the first phase, one "seeder" thread traverses the upper k levels of the BVH and records the node IDs of all the leaf nodes encountered and the node IDs of the k'th level subtrees. Though no other thread can start refitting until this seeding is done, there is no scalability issue as the seeding has to traverse only a very small number of nodes, is thus extremely cheap, and so can be run by one of the update threads before the thread continues with its load balanced vertex updating. This ensures that the seeding cost is load balanced with the vertex updates.

Once all the $N-1$ update threads are done updating the vertices and seeding, they synchronize on a barrier, and then switch to BVH refitting. We dynamically load-balance by having each thread take a node ID from the list, refit that subtree, and repeat. As soon as a thread finds no more subtrees to refit, it immediately goes on to performing triangle updates. The last thread to *finish* a subtree update also performs the final "merge" of the refit subtrees.

Triangle Update For ray-triangle intersection, we use the method outlined in [Wald 2004]. This method uses a precomputed set of data values for each triangle, which for an animated scene has to be recomputed every frame. Due to imbalances in the BVH refitting phase (i.e., the last thread having to merge the subtrees), the update threads can enter that phase at different times. We compensate by dynamically load balancing the triangle updates. In this way, all of the individual operations—vertex update, serial subtree seed, parallel subtree update, serial subtree merge, and triangle update—are fully interleaved, ensuring that all CPUs remain constantly utilized and finish at the same time.

Parallel Rendering Once all $N-1$ render threads have finished updating, they synchronize themselves via a barrier, and then render the scene using a standard tile-based dynamic load balancing scheme, as used by Wald [2004; 2006a].

Note that the entire per-frame rendering phase—update and render—is dynamically load-balanced at all stages, uses all $N-1$ threads all the time, and performs only three barrier operations per frame: after vertex updates, after all threads have completed updating, and once all tiles have been rendered. The only non-parallelizable stage is the time between the end of the current and the start of the next frame, in which the application processes user input, displays the image, and if applicable, swaps the rebuild data.

3.3 BVH Build Method

The choice of BVH build method is orthogonal to our approach, allowing any build method to be used with asynchronous rebuilds. Since building a BVH over multiple frames is explicitly allowed in our framework, one could in principle use very costly BVH builds

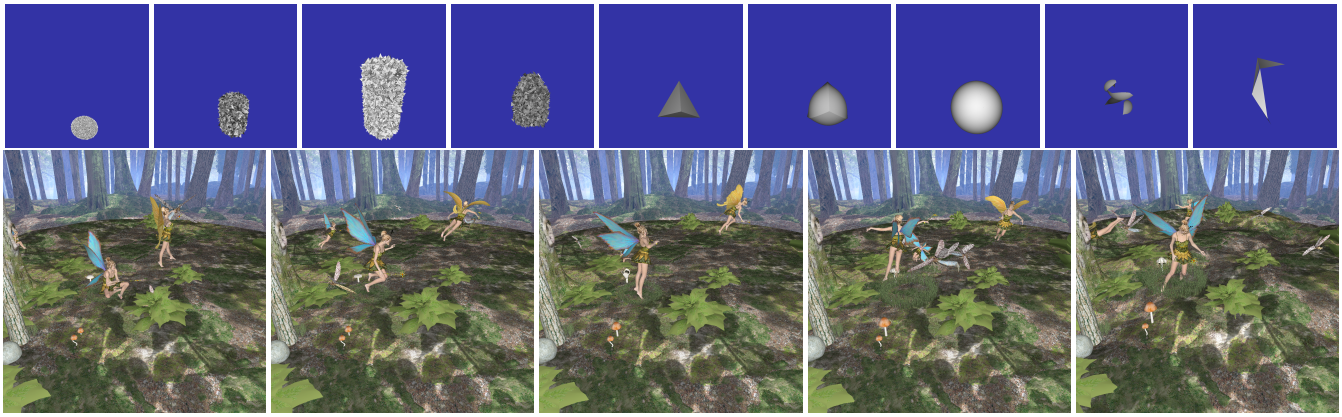

Figure 3: Five frames each from the "BART museum" and "Fairy Forest 2" scenes used in our experiments (using t=0s,3.125s, 6.25s, 9.375s, 12.5s, 15.625s 18.75s, 21.875s and 25s for the museum scene, and t=0s, 8s, 16s, 24s, and 31s for the Fairy scene, respectively).

that try to achieve the best possible BVH quality. However, when building asynchronously, a BVH will *always* be outdated by as many frames as it took to build this BVH. Thus, there is still a trade-off between a build method's resulting BVH quality and the time to achieve that quality, as longer builds potentially suffer worse from deterioration. We currently support the N log N sweep method outlined in [Wald et al. 2006a] and a centroid-based spatial median method in the spirit of [Wächter and Keller 2006]. By default we use the very fast to build centroid-based spatial median method.

4 Results and Discussion

For our experiments we use a 4 processor, dual core Opteron 870 PC with 16GB of RAM. Unless otherwise noted, we use all 8 cores in our tests. We use two test scenes: the "Fairy Forest 2" and the "BART museum" (see Figure 1). The Fairy Forest 2 has 394K triangles, most of which are deforming every frame, and resembles a game-like scene. The 64K triangle BART museum is part of the BART benchmark [Lext et al. 2000] and is intentionally designed to stress test large deformations. Because it deforms heavily by morphing into wildly varying shapes (see Figure 3), it provides a challenge where standard BVH refitting quickly breaks down (see [Wald et al. 2006a; Lauterbach et al. 2006]). All measurements were performed using the packet/frustum ray tracer used in [Wald et al. 2006a] at 1024×1024 pixels, with simple lambertian shading and no textures. We do not use shadows or other secondary rays, such as reflection, refraction, and so on; these would not influence scalability or build times at all, and would only affect render times.

Using the centroid-based spatial median build method on a single Opteron core, we can build a new BVH in roughly 25ms for the BART scene, and in roughly 225 ms for the Fair Forest 2 scene.

Deterioration and Disruptions In Figure 4 we compare our method with the standard "refit only" method which uses no rebuilds, and with Lauterbach's rebuild heuristic.

Both scenes show that simply refitting leads to severe performance deterioration, with about a $3\times$ drop for the Fairy scene, and a nearly complete standstill for the BART scene. Lauterbach's approach is clearly superior to refitting only; it avoids the BART scene's extreme deterioration, and achieves consistently higher frame rates for the Fairy scene. Furthermore, with only three rebuilds triggered for the Fairy scene, it achieves a nearly constant frame rate that is up to $2.5\times$ that of the refitting only approach.

While these experiments confirm Lauterbach's method is superior over deforming only, they also show its weaknesses: the high variation in frame rate caused by rebuilds and varying rates of deterioration, the "sawtooth" effect of deterioration until a new build is triggered, and in particular, the disruptions in which the system freezes until a new BVH is ready. That last effect is particularly visible in the fairy scene, where the otherwise nearly constant frame rate of 20-25Hz is unexpectedly interrupted three times, during which the system freezes for roughly 5 ordinary frames.

Compared to Lauterbach's method, our approach does not achieve the same peak performance, as we consistently "lose" one CPU to building BVHs, but we are generally quite close to it. In addition, our method achieves a significantly smoother frame rate without disruptions, and with a significantly reduced sawtooth effect.

Impact of Number of Cores While continuously rebuilding the BVH results in a faster to traverse BVH compared to a refitted BVH, the downside is that one core is always busy with building BVHs and cannot contribute to rendering. Thus, we consis-

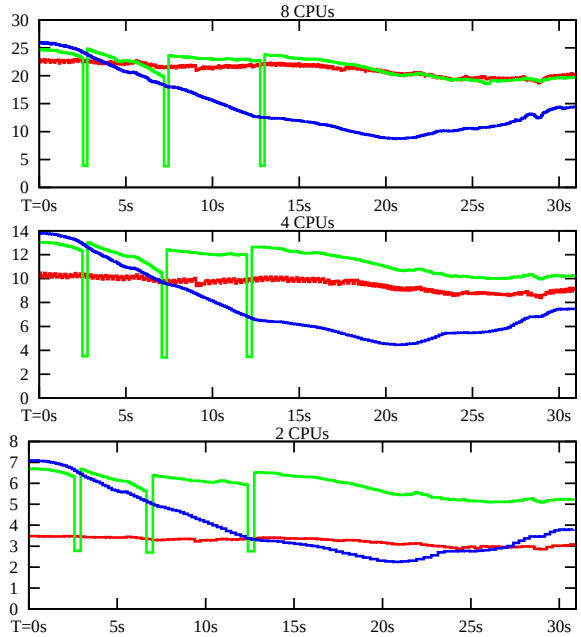

Figure 5: As the number of available CPUs increases, the advantage of asynchronous rebuilds increases.

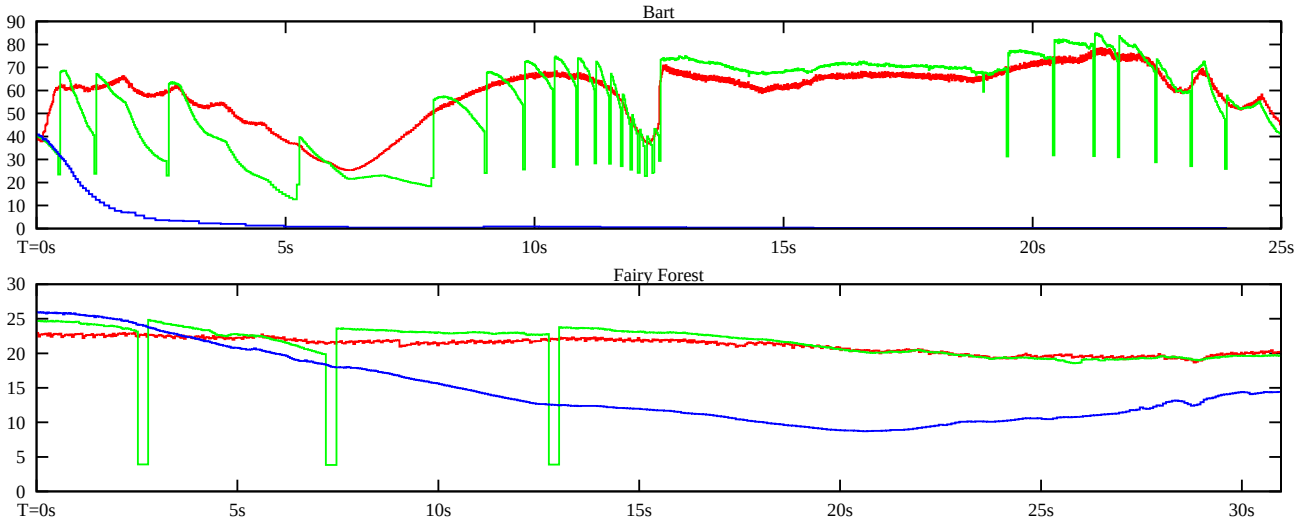

Figure 4: Impact on rendering performance for the three BVH build/update strategies: update only (blue), using Lauterbach et al.'s update heuristic (green), and using our asynchronous rebuilding strategy (red). Data is given for the "Fairy Forest 2" (394k triangles, 31sec animation) and the "BART museum" (66K triangles, 25sec animation), and are measured on a 4 processor dual-core Opteron PC. Since updating only leads to performance deterioration, both our and Lauterbach's approach work significantly better than updating only. Compared to Lauterbach's approach, we achieve slightly inferior peak performance (since we always spend some time on rebuilding), but maintain much more stable frame rates, and in particular, avoid the disturbing "freezes" that occur whenever Lauterbach's rebuild heuristic triggers a rebuild.

tently have one thread less for rendering than when using the rebuild heuristic or refitting only. As expected, Figure 5 shows that this effect is particularly severe for two cores (in which case we spend half our compute potential on BVH construction), but diminishes for more cores. Increasing the number of CPUs used by our method reduces the fraction of CPU time spent on rebuilding, and thus reduces our method's overhead.

Interestingly, even for the worst case of 2 threads our *worst* frame time is still better than that of either Lauterbach or refitting only, even if the average frame time is far worse. For 8 cores, this overhead is nearly gone; while one would expect our peak performance to be $1/8^{th}$ lower than the rebuild heuristic's performance, in practice the difference is much smaller due to the overhead in computing the rebuild heuristic.

When adding more cores, the frame rate will improve, but the build time will not. Consequently, we will render more frames during one build cycle, and have to deform the existing BVH more often before a new one is available. Though this might seem problematic at first, it is not in practice. Virtually all interactive animations, such as games, operate in world time, and a higher frame rate simply means a smoother animation by taking smaller timesteps and rendering the same animation with more frames. As such, the deformation accumulated by the time the new BVH is ready is independent of the number of cores or frame rate.

Impact of Build Method and Build Time As mentioned in Section 3.3, a better build method will result in a higher quality BVH, which in principle will translate to higher render performance; but, the longer to build method also means that the BVH will have degraded more by the time it is ready to be used and will degrade even more while the next BVH is being built, potentially leading to severe degradation.

To quantify that effect we have run our two animation sequences with two different build methods: a very fast approximate build as described by Wächter et al. [2006], and the SAH sweep method outlined in [Wald et al. 2006a]. For our two scenes, the sweep method's BVHs usually had a $\approx 1.5\times$ lower expected traversal cost, for the frame it was built for, but took roughly $18\times$ as long to build.

As can be see in Figure 6, both of the above mentioned effects are clearly visible in form of a "sawtooth" pattern in frame rate. The sweep method does reach a higher peak performance at the beginning, and when the rate of deformations is low, but as the deformation rate increases the slow to build sweep method clearly performs worse. This clearly argues for the faster to build methods, even if the build can be done asynchronously.

4.1 Practicability for Future Ray Tracing Systems

While the previous results have shown that our approach can indeed produce better results than current approaches based on either

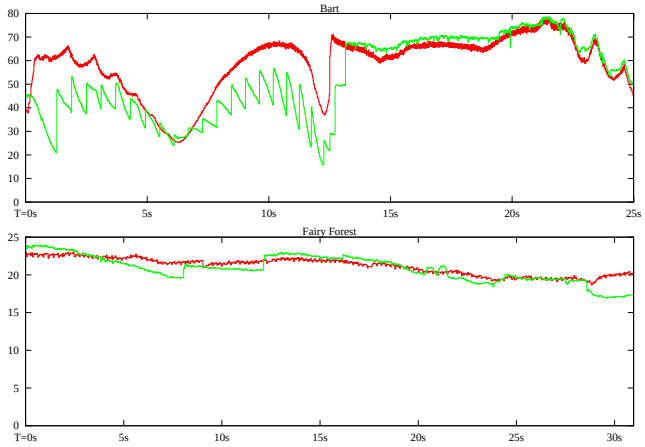

Figure 6: High-quality sweep-build (green) vs fast approximate build (red). The sweep build generates better BVH quality, but takes significantly longer to build. Due to the longer build times, the sweep build exhibits a "sawtooth" effect: longer builds imply longer times of deformation, leading to noticeable degradation over time in the BART scene. The sweep build sometimes even produces *lower* frame rates than the approximate build, since its BVHs are already significantly outdated by the time they are finished building.

refitting or rebuild heuristics, its practicability for future ray tracing applications will depend on how adaptable it is to different hardware architectures, to the growing demands on scene size, types of animation, and render quality.

Higher per-core performance A potentially higher performance per core (e.g., through a higher clock rate) would increase the frame rate, but would also result in higher build performance. Thus, as argued in the previous paragraph, the *absolute* time we have to rely on deformation will go down, effectively increasing the practicability of our method.

Impact of render cost per pixel Just like changing the number of cores, by increasing the cost per pixel (e.g., by tracing more rays for advanced effects) we merely change the ratio of build time to render time, and has the same effect as reducing the number of cores: the frame rate goes down, but the absolute time we have to rely on deforming an old BVH isn't affected. In fact, the argument can be reversed: if future ray tracers will spend more rays per pixel, and if future chips will have more cores, then we can use the additional cores for tracing more rays per pixel at the same frame rate, and without negatively affecting the dependence on deformation at all.

Scene Complexity and Animation Speed While most current games do not use more triangles than our 394K triangle Fairy Forest 2 scene, if significantly larger scenes were used, the $O(N \log N)$ build method would require that we rely longer on deformation before a new BVH is available. This, and increasing the animation speed, is similar in spirit to using a slower to build BVH as mentioned above and would share the same results. Furthermore, if a scenegraph is available, which is often the case in games, the scene could be easily decomposed into subsets which could each be rebuilt asynchronously on separate cores. On the other hand, many researchers argue that future games might make heavy use of freeform or subdivision geometry, and would therefore need significantly less primitives than are used today [Stoll et al. 2006]. In that case, our build times would shrink as well.

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

Similarly, changing scene topology is currently not supported. However, most scenes can be structured to avoid topology changes, for example, by translating a character out of or into a scene, and when that is not possible, it might be possible to predict the need for a topology change and asynchronously build the new BVH in another thread.

Application to non-CPU architectures Though we have only talked about standard multi-core CPU architectures so far, our method is also applicable to other architectures. On a CELL [Benthin et al. 2006], for example, the rebuild could happen on the PowerPC core, with the render threads running on the SPEs. Even more interesting, the method could also be used for ray tracers running on GPUs, or for special-purpose ray tracing hardware as proposed by Woop et al. [2006]. For these architectures, rendering and BVH updating can easily be performed in parallel on the respective hardware architectures [Woop et al. 2006], but rebuilding doesn't easily map to such architectures. Using our method, the update would run on the respective SPE, GPU, or RPU, while the rebuild is asynchronously performed on the host CPU.

5 Conclusion

In this paper, we have presented a new approach to handling dynamic scenes in a highly parallel ray tracing architecture. Instead of trying to do a full BVH rebuild per frame, we avoid any kind of scalability issues by rebuilding asynchronously over the course of multiple frames, and in the meantime rely on refitting, which parallelizes quite well.

The method is particularly designed for highly parallel architectures with multiple parallel cores, be it CPU cores, GPU cores, CELL SPE, or even special purpose hardware. While increasing parallelism is a problem for pure rebuilding, our method in fact benefits from more cores, as the relative overhead decreases. As argued in the previous section, the currently foreseeable trends towards having many more cores, slightly more performance per core, and more rays per pixel do not negatively affect our method's practicability.

Our method's advantage over existing methods depends on the scene, and on the amount of deformation in a scene. If the deformation is sufficiently small, simply refitting every frame may suffice, rendering our method superfluous; the same is true if the scene is sufficiently small to be rebuilt per frame. Compared to Lauterbach's update heuristic, we usually achieve a somewhat lower *average* performance (due to rebuild overhead), but avoid performance degradations and system response disruptions, which is important for truly interactive applications like games.

Since we still depend on a scene's deformability for at least short periods of time, we cannot currently handle randomly deforming scenes, or scenes with changing topology; for such severe scenes, another data structure such as Wald's "Coherent Grid Traversal" [2006b] may be more applicable.

In future work, we will look into different BVH build methods that offer a good trade-off between build time and BVH quality. More importantly, we would like to see our framework applied to systems like the RPU, or to a CELL based ray tracer. Ideally, this would happen in a truly dynamic environment, such as in a real game.

References

BENTHIN, C., WALD, I., SCHERBAUM, M., AND FRIEDRICH, H. 2006. Ray Tracing on the CELL Processor. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*.

GREEN, S. A., AND PADDON, D. J. 1990. A Highly Flexible Multiprocessor Solution for Ray Tracing. *The Visual Computer 6*, 2, 62–73.

GÜNTHER, J., FRIEDRICH, H., WALD, I., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Ray Tracing Animated Scenes using Motion Decomposition. In *Proceedings of Eurographics*.

HAVRAN, V., HERZOG, R., AND SEIDEL, H.-P. 2006. On Fast Construction of Spatial Hierarchies for Ray Tracing. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*.

HUNT, W., STOLL, G., AND MARK, W. 2006. Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*.

INTEL, 2006. http://www.intel.com/go/terascale/.

IZE, T., WALD, I., ROBERTSON, C., AND PARKER, S. G. 2006. An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, I. Wald and S. G. Parker, Eds., 47–55.

LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*.

LEXT, J., ASSARSSON, U., AND MÖLLER, T. 2000. BART: A benchmark for animated ray tracing. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, May.

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

PARKER, S. G., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B. E., AND HANSEN, C. D. 1999. Interactive ray tracing. In *Proceedings of Interactive 3D Graphics*, 119–126.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics 24*, 3, 1176–1185. (Proceedings of ACM SIGGRAPH 2005).

STOLL, G., MARK, W. R., DJEU, P., WANG, R., AND ELHASSAN, I. 2006. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Tech. Rep. 06-21, University of Texas at Austin Dep. of Comp. Science.

WÄCHTER, C., AND KELLER, A. 2006. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Proceedings of the 17th Eurographics Symposium on Rendering*.

WALD, I., AND HAVRAN, V. 2006. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3, 153–164. (Proceedings of Eurographics).

WALD, I., BOULOS, S., AND SHIRLEY, P. 2006. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics (conditionally accepted)*. Available as SCI Institute, University of Utah Tech.Rep. UUSCI-2006-023.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics 25*, 3, 485–493. (Proceedings of ACM SIGGRAPH 2006).

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University.

WATSON, B., WALKER, N., RIBARSKY, W., AND SPAULDING, V. 1998. Effects of Variation in System Responsiveness on User Performance in Virtual Environments. *Human Factors 40*, 3, 403–404.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: A programmable ray processing unit for realtime ray tracing. In *Proceedings of SIGGRAPH*, 434–444.

WOOP, S., MARMITT, G., AND SLUSALLEK, P. 2006. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*.