Eurographics Symposium on Parallel Graphics and Visualization (2007)
Jean M. Favre, Luis Paulo dos Santos, and Dirk Reiners (Editors)

Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures

Thiago Ize Ingo Wald Steven G. Parker

SCI Institute, University of Utah, USA

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.

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 providing around 330 GFlops and increasing programmability [Lue06]. 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 already has an 80 core processor prototype capable of TeraFlop performance [Int06].

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 easily fulfill the growing quality demands, such as soft shadows, depth-of-field, caustics, participating media, and global illumination; but the main limitation is that it is not yet efficient enough for use in dynamic applications such as games. However, 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 [GFW*06, HSM06], grids [WIK*06], or bounding volume hierarchies (BVHs) [LYTM06, WBS07], but there are trade-offs associated with each of these data structures. Kd-trees seem to offer the highest ray tracing performance, but are most costly to build [WH06]; grids are efficient to build [WIK*06], 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 [WBS07]. While one could rebuild a deteriorated BVH to restore performance, this creates a disruptive pause while the BVH is being rebuilt [LYTM06].

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 handles scenes with large deformations over time. In particular, our approach is especially suited for the highly parallel, multi-core architectures as are currently foreseeable for the near future. Instead of alternating between phases of rendering and (potentially infrequent) building, our approach exploits the parallelism inherent in a multi-core system by continuously and *asynchronously* rebuilding new BVHs (potentially over the course of multiple frames) while the remaining cores concurrently update and traverse the most recently built 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 [GP90, Muu95, PMS*99]. With the growing capabilities of commodity architectures, researchers then turned their attention to ray tracing on PCs, and PC clusters [WSBW01]. In particular since Reshetov's "Multilevel Ray Traversal" [RSH05], 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 that same year, Woop et al. [WSS05] 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 efficient spatial acceleration structures, such as a BVH, kd-tree, or grid. While there has 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. [GFW*06] 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. [SMD*06] proposed a lazy build mechanism that is aided by scene-graph information, but no real-time data is yet available. For rendering general dynamic scenes with a kd-tree, Popov et al. [PGSS06] and Hunt et al. [HSM06] have investigated fast approximate methods to rebuild kd-trees from scratch; these, however, are still rather slow, and seem to parallelize poorly ([PGSS06] 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. [WIK*06] 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. [IWRP06] 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 triangles, BVHs build an *object hierarchy*, and in each tree node store a bounding volume for that subtree's geometry. Wald et al. [WBS07] 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. [LYTM06]. Other BVH inspired approaches have been proposed by Havran et al. [HHS06], Woop et al. [WMS06], and Wächter et al. [WK06].

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 unchanged and only refit the BVH nodes' bounding volumes. While this refitted BVH may have a different and potentially less efficient tree structure than one built from scratch, the refitted BVH will nonetheless 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 triangle count or 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. [LYTM06], 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. [LYTM06] 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 [WWRS98].

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 rebuilding and incremental updating do not scale as well as 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 few 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 1).

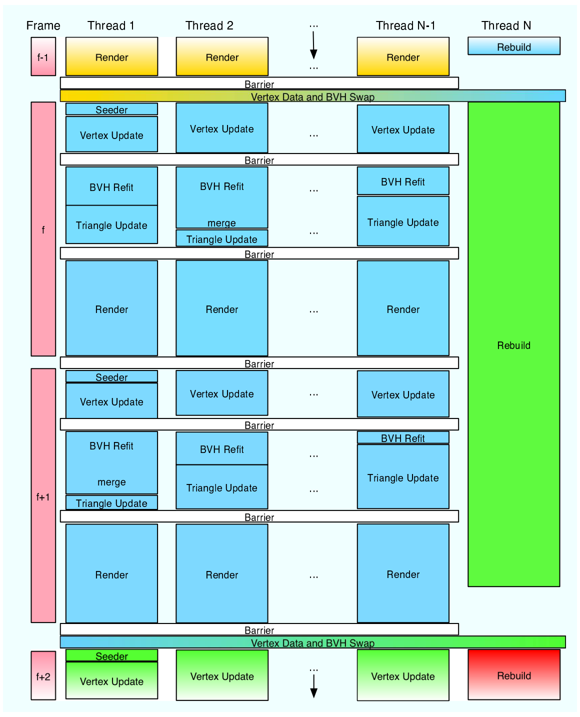

Figure 1: *Given a highly parallel architecture with N cores, $N-1$ of the cores work on parallel rendering and BVH refitting of the most recently finished BVH, while the N^{th} core works asynchronously and builds new BVHs as fast as possible, potentially over multiple frames (2 in this example). BVHs are deformed for only a few frames, and both scalability bottlenecks and pauses are avoided altogether.*

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 80 bytes extra storage per triangle, which for most scenes has a minor impact. A large 1M triangle scene, for instance, would only require roughly 80MB of extra storage.

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. Since at that time the application has not yet computed the new vertex positions, we start the build process with vertex positions that are already one frame outdated. 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, by building the BVH from the last finished frame's vertex positions, the vertex and BVH buffers can be switched quickly with two pointer swaps, and the builder, application, and render threads can immediately continue. Furthermore, since it will likely take several frames before the new BVH is available anyways, that BVH will already 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 ("hot swapping"), it would require extra synchronization during rendering as well as refitting the newly completed BVH to the current frame being rendered. These issues 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 needed by all following 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.

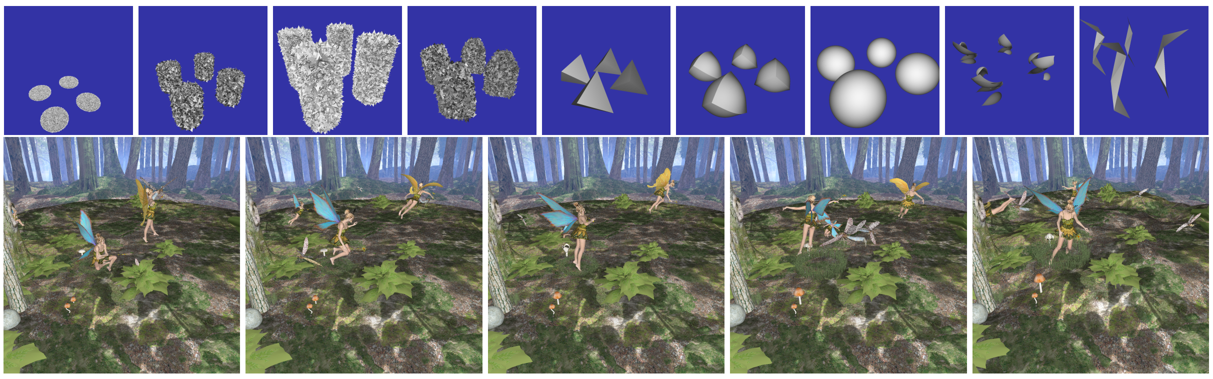
Figure 2: *Frames from the "BART museum" and "Fairy Forest 2" scenes used in our experiments (using t=0s, 1s, 2s, 3s, 4s, 5s, 6s, 7s, and 8s for the museum scene, and t=0s, 2s, 4s, 6s, and 7.75s for the Fairy scene, respectively).*

Parallel BVH Refit Once the new vertex positions are known, we 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. [LYTM06] 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 that 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 [Wal04]. 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—parallel subtree update, serial subtree merge, and triangle update—are fully interleaved, ensuring that all cores 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 [Wal04].

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 our method to be used with any build method, including incremental update methods, if required. Since building a BVH over multiple frames is explicitly allowed in our framework, one could in principle use very costly BVH builds 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 $O(N \log N)$ sweep method outlined in [WBS07] and a centroid-based spatial median method in the spirit of [WK06]. By default we use the very fast to build centroid-based spatial median method.

4. Results and Discussion

Although mainly designed for upcoming multi-core architectures that will likely contain a large number of cores, current processor architectures only feature 2 to at most 4 cores per processor. We therefore use an 8 processor dual-core Opteron 880 shared-memory PC for our experiments, which we believe most closely resembles our target processor designs. Unless otherwise noted, we use all 16 cores in

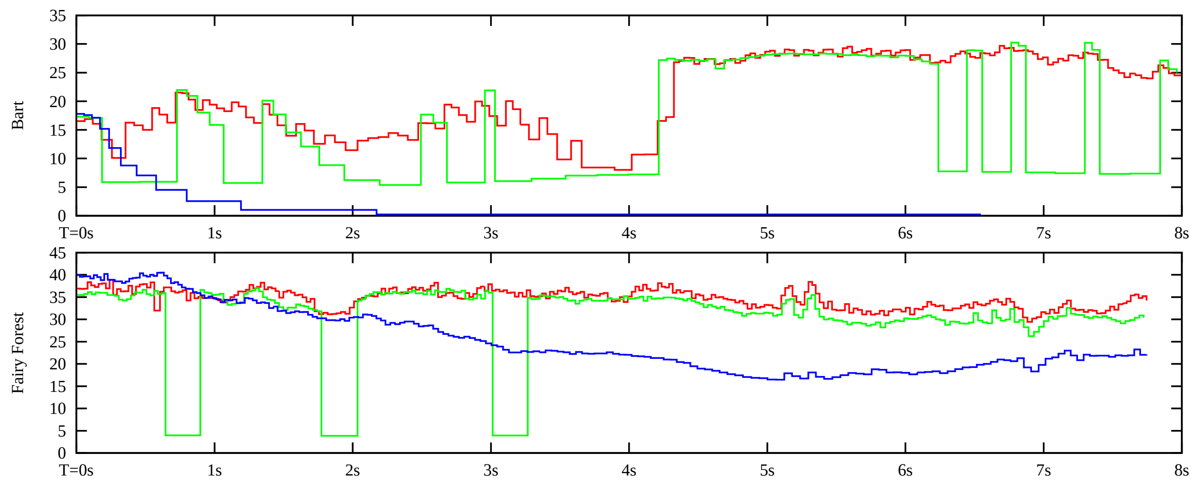

Figure 3: *Impact on frame rate for the three BVH build/update strategies: refit only (blue), using Lauterbach et al.'s update heuristic (green), and using our asynchronous rebuilding strategy (red). Data is given for the "Fairy Forest 2" (394k triangles) and the "BART museum" (262K triangles), and are measured on an 8 processor dual-core Opteron PC. Since refitting only leads to performance deterioration, both our and Lauterbach's approach work significantly better than refitting only. Compared to Lauterbach's approach, we achieve roughly equal peak performance, but maintain much more stable frame rates, and in particular, avoid the disturbing "freezes" that occur whenever Lauterbach's rebuild heuristic triggers a rebuild.*

our tests. We use two test scenes: the "Fairy Forest 2" and the "2×2 BART museum" (see Figure 2). The Fairy Forest 2 is a 7.75s long keyframed animation with 394K triangles, almost all of which are deforming every frame, and resembles a game-like scene. The 262K triangle 2×2 BART museum is 8s long and composed of 4 copies of the museum scene from the BART benchmark [LAM00] and is intentionally designed to stress test large deformations. Because it deforms heavily by morphing into wildly varying shapes (see Figure 2), it provides a challenge where standard BVH refitting quickly breaks down (see [LYTM06, WBS07]). All measurements were performed using the packet/frustum ray tracer used in [WBS07] 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, would only affect render times, and are therefore completely orthogonal to our approach.

Using the centroid-based spatial median build method on a single Opteron core, we can build a new BVH in roughly 170ms for the BART scene, and in roughly 230 ms for the Fairy Forest 2 scene.

Parallel building Using more than one core for the build would be an obvious and straightforward extension, and would fit well within our approach. Since building is asynchronous to rendering, perfect scalability of the build process would not even be required, as any "idle" cycles during the build could be used by the render threads. Though the benefit of a parallel rebuild for moderately sized scenes running on a small number of cores is not clear, for future systems with many cores, such as Intel's 80-core prototype [Int06], a parallel build would likely be beneficial.

Deterioration and Disruptions In Figure 3 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 $2.3\times$ 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 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 $2\times$ 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 temporarily freezes while a new BVH is being built. That last effect is particularly visible in the fairy scene, where the otherwise nearly constant frame rate of 30-35Hz is unexpectedly interrupted three times, during which the system freezes for roughly 9 ordinary frames.

Compared to Lauterbach's method, our approach actually consistently performs better than Lauterbach's method, despite rendering with one less core. This is due to the overhead inherent in calculating a rebuild heuristic being more expensive than the loss of 1 out of 16 cores for rendering. Furthermore, our method achieves a significantly smoother frame rate without disruptions, and with a significantly reduced sawtooth effect.

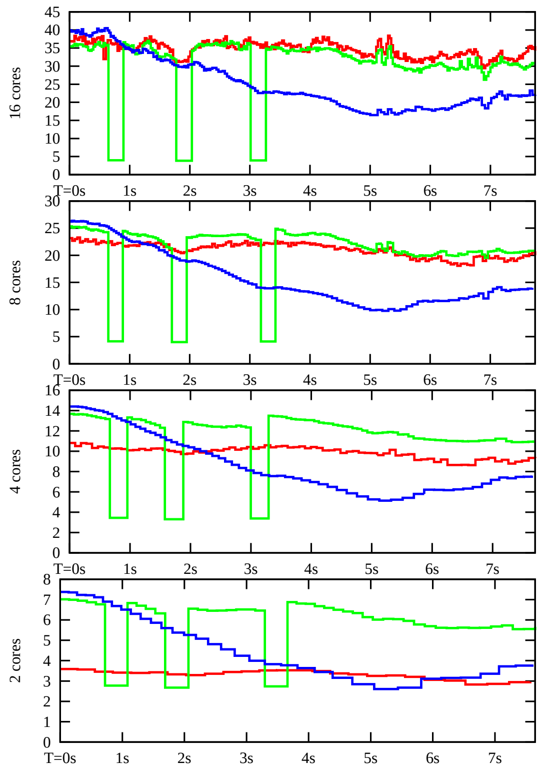

Figure 4: *As the number of available cores increases, the advantage of asynchronous rebuilds (red) increases over both the rebuild heuristic (green) and refit only (blue), as seen in the frame rate for the Fairy Forest 2 scene.*

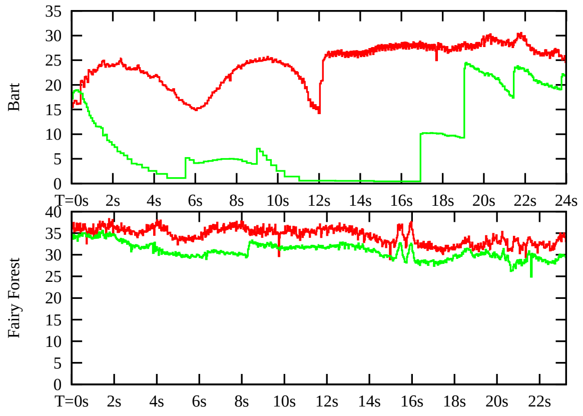

Figure 5: *Frame rate for high-quality sweep-build (green) vs fast approximate build (red). The sweep build generates better BVH quality, but takes significantly longer to build. Due to the longer build times, the sweep build exhibits a "sawtooth" effect: longer builds imply longer times of deformation, leading to noticeable degradation over time in the BART scene. The sweep build even produces far* lower *frame rates than the approximate build, since its BVHs are already significantly outdated by the time they are finished building. We run these animations at 3× slower speed so that the sweep build has a chance to occur multiple times.*

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 consistently have one thread less for rendering than when using the rebuild heuristic or refitting only. As expected, Figure 4 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 cores 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 $\frac{7}{8}\times$ that of the rebuild heuristic, in practice the difference is much smaller due to the overhead in computing the rebuild heuristic, and in fact, by 16 cores we perform better than 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 having a larger budget in rays that can be traced per second simply means a smoother animation by taking smaller timesteps and rendering the same animation with more frames, or rendering with more rays per pixel (i.e., at higher quality). 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. [WK06], and the SAH sweep method outlined in [WBS07]. For our two scenes, the sweep method's BVHs usually had around a 1.5× lower expected traversal cost, for the frame it was built for, but took roughly 18× as long to build.

As can be seen in Figure 5, both of the above mentioned effects are clearly visible in the form of a "sawtooth" pattern in frame rate and lower peak performance due to the sweep BVH already being heavily outdated by the time it is finally built. Because the sweep build takes so long to build, we ran these animations at one third the normal speed so that these effects would be visible. At the normal speed only two rebuilds are accomplished during the entire animation, resulting in the performance almost matching that of just performing the refitting. This clearly argues for the faster to build or parallel 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 produce better results than current approaches based on either 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 both the frame rate, and the build performance. Thus, as previously argued, the *absolute* time we have to rely on deformation will go down, resulting in less BVH degradation, which increases 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, which 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 is not 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 fewer primitives than are used today [SMD*06]. In that case, our build times would shrink as well, thus further improving the practicability of our method.

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 exploding objects, 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, this usually occurs only if entire objects appear into/disappear from a composite scene environment, which can easily be handled by a two-level approach as proposed in [WBS03]: the small top-level scene could easily be rebuilt per frame, with our method handling the per-object deformations, potentially for multiple models in parallel.

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 [BWSF06], for example, the render threads could run on the SPEs, with the rebuild thread running on either one of the SPEs or on the slower but more flexible PowerPC core.

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. [WMS06]. For these architectures, rendering and BVH updating can easily be performed in parallel on the respective hardware architectures [WMS06], 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 system and is especially suitable for multi-core hardware architectures. 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 multi-core architectures, be it CPU cores, GPU cores, CELL SPEs, 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 would make our method even more suitable for these architectures than the one we used in our experiments.

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 (though any such approach might still suffer from scalability issues). Compared to Lauterbach's rebuild heuristic, for a small number of cores we achieve a lower *average* performance, but avoid performance degradations and system response disruptions, which is important for truly interactive applications like games. When more cores are available, we actually perform consistently better than the rebuild heuristic since we do not need to calculate the heuristic. This means that given enough cores, it is better to always perform asynchronous rebuilds, despite the loss of one core to rendering and updating, than trying to rebuild asynchronously only when the BVH efficiency is low.

Since we still depend on a scene's deformability for at least short periods of time, we cannot handle randomly deforming scenes, or scenes with changing topology; for such severe scenes, another data structure, such as Wald's "Coherent Grid Traversal" [WIK*06], 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, the CELL, or upcoming multi-core architectures. Ideally, this would happen in a truly dynamic environment, such as in a real game. Finally, in particular for architectures with many more cores, we would like to extend our method to incorporate parallel BVH building on a larger number of threads.

References

[BWSF06] BENTHIN C., WALD I., SCHERBAUM M., FRIEDRICH H.: Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 15–23.

[GFW*06] GÜNTHER J., FRIEDRICH H., WALD I., SEIDEL H.-P., SLUSALLEK P.: Ray Tracing Animated Scenes using Motion Decomposition. In *Proceedings of Eurographics* (2006).

[GP90] GREEN S. A., PADDON D. J.: A Highly Flexible Multiprocessor Solution for Ray Tracing. *The Visual Computer 6*, 2 (1990), 62–73.

[HHS06] HAVRAN V., HERZOG R., SEIDEL H.-P.: On Fast Construction of Spatial Hierarchies for Ray Tracing. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[HSM06] HUNT W., STOLL G., MARK W.: Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[Int06] INTEL: http://www.intel.com/go/terascale/, 2006.

[IWRP06] IZE T., WALD I., ROBERTSON C., PARKER S. G.: An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 47–55.

[LAM00] LEXT J., ASSARSSON U., MÖLLER T.: *BART: A Benchmark for Animated Ray Tracing*. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, May 2000.

[Lue06] LUEBKE D.: Introduction. In *Supercomputing Tutorial on GPGPU* (2006), Harris M., Luebke D., (Eds.).

[LYTM06] LAUTERBACH C., YOON S.-E., TUFT D., MANOCHA D.: RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 39–45.

[Muu95] MUUSS M.: Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium* (1995).

[PGSS06] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[PMS*99] PARKER S. G., MARTIN W., SLOAN P.-P. J., SHIRLEY P., SMITS B. E., HANSEN C. D.: Interactive ray tracing. In *Proceedings of Interactive 3D Graphics* (1999), pp. 119–126.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics 24*, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH 2005).

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

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

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

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (2007).

[WH06] WALD I., HAVRAN V.: 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* (2006).

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493. (Proceedings of ACM SIGGRAPH 2006).

[WK06] WÄCHTER C., KELLER A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Proceedings of the 17th Eurographics Symposium on Rendering* (2006).

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware* (2006).

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics).

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A programmable ray processing unit for realtime ray tracing. In *Proceedings of SIGGRAPH* (2005), pp. 434–444.

[WWRS98] WATSON B., WALKER N., RIBARSKY W., SPAULDING V.: Effects of Variation in System Responsiveness on User Performance in Virtual Environments. *Human Factors 40*, 3 (1998), 403–404.

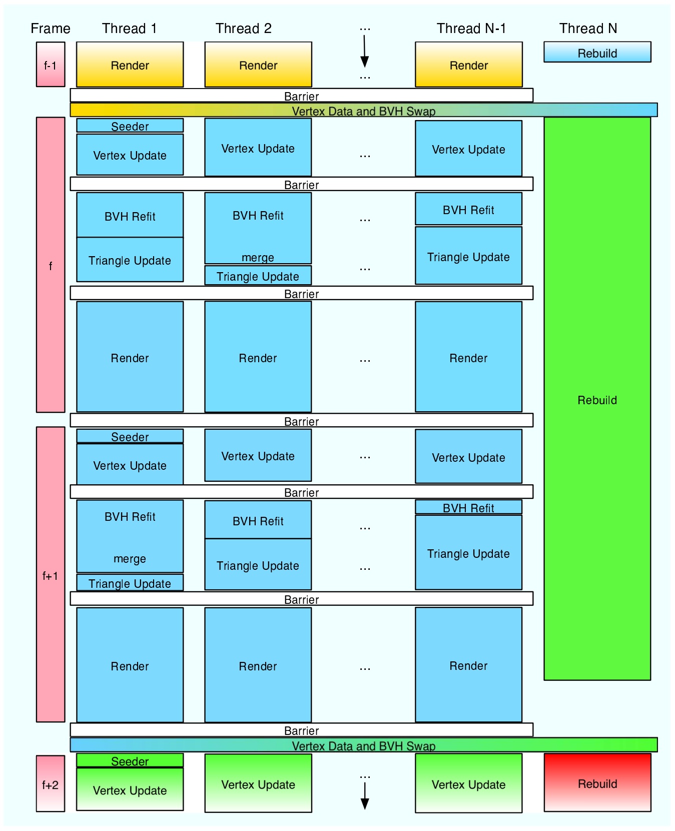

Figure 1: *Given a highly parallel architecture with N cores, $N-1$ of the cores work on parallel rendering and BVH refitting of the most recently finished BVH, while the N^{th} core works asynchronously and builds new BVHs as fast as possible, potentially over multiple frames (2 in this example). BVHs are deformed for only a few frames, and both scalability bottlenecks and pauses are avoided altogether.*