SCI INSTITUTE
TECHNICAL REPORT

Ray Tracing Animated Scenes using Coherent Grid Traversal

Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker

UUSCI-2006-014

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

April 5, 2006

Abstract:

We present a new approach to interactive ray tracing of animated scenes based on traversing frustum-bounded packets of coherent rays through uniform grids. By incrementally computing the overlap of the frustum with a slice of grid cells, we accelerate grid traversal by more than a factor of 10, and achieve ray tracing performance competitive with the fastest known packet-based kd-tree ray tracers. The ability to efficiently rebuild the grid on every frame enables this performance even for fully dynamic scenes that typically challenge interactive ray tracing systems.

THE UNIVERSITY OF UTAH

Ray Tracing Animated Scenes using Coherent Grid Traversal

Ingo Wald Thiago Ize Andrew Kensler Aaron Knoll Steven G. Parker
SCI Institute, University of Utah

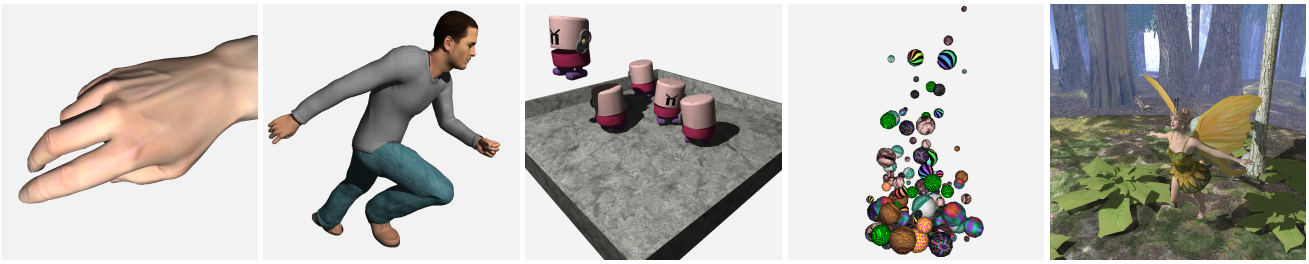

Figure 1: Several animated models ray traced using our coherent grid traversal: a) A gesturing hand of 15K triangles. b) An animated "Poser" model (78K triangles). c) Animated wind-up toys (11K triangles) walking incoherently around among each other. d) A rigid-body dynamics simulation of marbles (8.8K triangles). e) A complex scene of 174K animated triangles, where a fairy and a dragonfly dancing through an animated forest. Scenes are rebuilt from scratch every frame, allowing fully dynamic animation. Excluding shadows and shading, we can rebuild and ray trace these scenes at 1024×1024 pixels at 29.5, 13.3, 25.8, 47.8, and 3.1 frames per second on a dual 3.2 GHz Xeon.

Abstract

We present a new approach to interactive ray tracing of animated scenes based on traversing frustum-bounded packets of coherent rays through uniform grids. By incrementally computing the overlap of the frustum with a slice of grid cells, we accelerate grid traversal by more than a factor of 10, and achieve ray tracing performance competitive with the fastest known packet-based kd-tree ray tracers. The ability to efficiently rebuild the grid on every frame enables this performance even for fully dynamic scenes that typically challenge interactive ray tracing systems.

1 Introduction and Related Work

Over the last 20 years, a number of different data structures have been proposed for accelerating ray tracing, such as Bounding Volume Hierarchies (BVH), Grids, Octrees [Glassner 1984], and Binary Space Partitioning (see, e.g., [Glassner 1989; Havran 2001] for an overview). Each of these data structures has their own strengths and weaknesses, and the effectiveness of each technique strongly depends on the scene, application, and efficiency of the actual implementation. Recent work in interactive ray tracing, however, has focused primarily on kd-trees [Wald 2004; Foley and Sugerman 2005; Reshetov et al. 2005; Woop et al. 2005] on multilevel grids [Parker et al. 1999b; Reinhard et al. 2000; Purcell et al. 2002].

While the first interactive ray tracers used grids [Parker et al. 1999b], algorithmic developments for kd-tree based ray tracers – most notably coherent ray tracing [Wald et al. 2001] and MLRT traversal [Reshetov et al. 2005] – have significantly improved the performance of kd-trees. Packet tracing [Wald et al. 2001] creates groups of spatially coherent rays that are simultaneously traced *together* through a kd-tree, where all rays perform each traversal iteration in lock-step. This enables effective use of SIMD extensions on modern CPUs, increases the computational density of the code, and reduces strain on memory access. In turn, this gives rise to fast software implementations [Wald et al. 2001], and instruction-parallel design of special-purpose ray tracing hardware [Woop et al. 2005]. Exploiting the coherence in a packet of rays has yielded further improvements in "Multilevel Ray Tracing" (MLRT) [Reshetov et al. 2005], where a bounding frustum drives the kd-tree traversal of rays in bulk instead of considering each ray individually. Consequently, the cost of a traversal step becomes independent of the number of rays in the packet, encouraging larger packets with significantly lower cost per ray.

Unfortunately, these techniques are not directly applicable to grids. Thus, packet-enabled kd-trees have recently shown to consistently outperform grid-based ray tracers, and many believe that they are a superior acceleration structure for most interactive applications (see, e.g., [Stoll 2005]).

Dynamic Scenes Although packet kd-tree traversals outperform grids for static scenes, animated scenes present a challenge due to the high computational cost of rebuilding a kd-tree as objects move. For the surface area heuristics required to build fast kd-trees [Reshetov et al. 2005], building the acceleration structure effectively requires seconds to minutes for moderately complex scenes. This limitation to static scenes limits the utility of interactive ray tracing for many applications that would benefit from advanced lighting models, such as visual simulation, animations, and interactive games. While some efforts have focused on extending kd-trees to dynamic scenes [Wald 2004; Wald et al. 2003], they are limited to simple hierarchical motion of rigid bodies, and therefore are unsuitable for most truly dynamic animations that require unstructured motion. For full generality, we propose rebuilding the acceleration structure from scratch every frame. With kd-trees, this is currently infeasible.

A grid, in contrast, can be created and modified at interactive rates [Reinhard et al. 2000]. Consequently, grids are attractive for dynamic scenes because of their faster build, even if they have a higher traversal cost than a kd-tree. Nevertheless, as kd-trees can be up to an order of magnitude faster than single-ray grids, grids will only be viable when their traversal can be performed with similar efficiency. Ultimately, this will require employing the same techniques for grids that made kd-trees as fast as they are today: coherent packets of rays, SIMD, and frusta. However, the 3D digital differential analyzer (3DDDA) algorithms usually used for traversing a grid do not lend well to packetization, as we will explain below.

In this paper, we propose a new traversal scheme for grid-based acceleration data structures that allows for traversing and intersecting packets of coherent rays using an MLRT-inspired frustum-traversal scheme. This algorithm is well-suited for SIMD implementation and provides dramatic speedup over a conventional grid traversal, yielding performance comparable to kd-tree based systems for static scenes. More importantly, this scheme facilitates animated scenes in a straightforward manner by interactively rebuilding the data structure from scratch every frame. Using this technique on fully animated scenes of up to about 100,000 triangles, we achieve a ray tracing performance of 10-50 frames per second (at 1024×1024 pixels) on a dual 3.2 GHz Xeon CPU.

2 Coherent Grid Traversal

Efficient ray-grid traversal has already received much attention [Cleary et al. 1983; Fujimoto et al. 1986; Amanatides and Woo 1987; Parker et al. 1999b; Spackman and Willis 1991], in aspects of both algorithm and implementation. Significant improvements cannot be expected from merely optimizing current implementations; we must explore new concepts to design an effective packetized traversal. Our new algorithm delivers to grids the same components that made KD-trees as fast as they are today: packets, SIMD extensions, and frustum traversal; while preserving the trivial computation of an incremental grid marching step.

In this section, we explain why these techniques have been successful for other acceleration structures and discuss the difficulties of applying the same concepts to a conventional grid traversal. Then, we derive our new packet traversal scheme, and show how it can benefit from known optimizations to achieve significantly higher performance than past grid implementations.

2.1 Issues with Packetized Grids

The basic idea of packet and frustum traversal is straightforward: rather than traverse each ray on its own, we exploit the intrinsic coherence between neighboring rays, and trace them together. If the rays are coherent, they will largely traverse the same regions of space, accessing identical nodes in an acceleration structure, and intersecting the same underlying triangles. Effectively, the cost of memory access becomes amortized over all the rays in a packet, ideally for both our acceleration structure and geometry data. In addition, traversing multiple rays through the same node of the acceleration structure allows us to perform SIMD operations on four rays at once, reducing the computation costs of both traversal and primitive intersection by up to a factor of four. Finally, frustum techniques determine intersection patterns of an entire packet, often replacing intensive per-ray branching with a single test.

The advantages of packets, SIMD and frustum methods are beneficial to any acceleration structure. Spatially hierarchical structures, such as a kd-tree or BVH, typically exhibit little divergence at the upper levels of traversal, making them ideally suited for adaptation to ray packets. Packets are easily traversed through hierarchical acceleration structures where rays generally progress through identical cells; diverging only in finer nodes deep down in the hierarchy, if at all. Even when rays diverge, some rays just traverse a few cells that they would not have traversed otherwise, but do not interfere with traversal decisions in the remaining part of the subtree. Since the packet is never divided, those rays automatically are re-enabled as soon as the recursion returns from that subtree.

For a grid, in contrast, the situation is more complicated: traversal is always performed on the same fine level, where divergence is most likely. Moreover, grid based ray tracers typically use 3D digital differential analyzers (3DDDA) or Bresenham-like algorithms to iterate through the voxels traversed by the ray (e.g., [Fujimoto et al. 1986; Amanatides and Woo 1987; Spackman 1990]). These algorithms can only chose one cell at a time to step into, but different rays can disagree on the next cell to be traversed. For example, Figure 2 shows four rays diverging in cell B; some demand traversal to C, while others demand traversal to D. If the packet decides to go to C first, the 3DDDA state variables for those rays entering cell D become invalid (and vice versa). These invalid state variables break the 3DDDA algorithm in the next traversal step.

This could be solved by splitting the packet into subpackets with the same traversal decision. However, Figure 2 shows that the rays that have diverged in cell A still traverse other common cells (E, F, G) later on. If the packet were split at cell B, that coherence would be lost. Re-merging the packets after each step would solve that problem, but is prohibitively expensive.

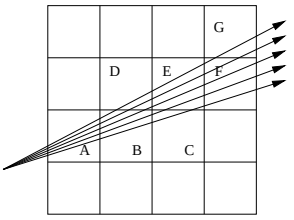

Figure 2: Four coherent rays traversing a grid. The rays are initially together in cells A and B, but then diverge at B where they disagree on whether to first traverse C or D in the next step. Even though they have diverged, they still visit common cells (E, F, G) afterwards.

2.2 A Slice-based Packet Traversal for Grids

As the above discussion has shown, the primary concern with packetizing a grid is that with a 3DDDA, different rays may demand different traversal orders. We solve this by abandoning 3DDDA altogether, and devise an algorithm that traverses the grid *slice by slice* rather than cell by cell. For example, we can traverse the rays in Figure 2 by traversing through vertical slices; from cell A in the first slice, we would traverse the rays to cells B and D in the second slice, then to C and E in the third, and so on. In each slice, we would intersect all rays with all of the slice's cells that are overlapped by at least one ray. This may traverse some rays through cells they would not have intersected themselves, but will keep the packet together at all times. In Figure 2, we would intersect 7 cells with 4 rays each, instead of 22 cell visits if the rays are traced individually. Though the packet now intersects only 7 instead of 22 cells, the total number of ray-cell intersection tests is $7 \times 4 = 28$. In practice, ray coherence easily compensates for this overhead.

We first transform the rays into the canonical grid coordinate system, in which a grid of $N_x \times N_y \times N_z$ cells maps to the 3D region of $[0..N_x) \times [0..N_y) \times [0..N_z)$. In that coordinate system, the cell coordinates of any $3D$ point p can be computed simply by truncating it. Then, in this coordinate system, we pick the dominant component (the $\pm X$, $\pm Y$ or $\pm Z$ axis) of the direction of the first ray. This will be the "major traversal axis" that we call $\vec{K}$; all rays are traversed along this same axis. The remaining dimensions are denoted $\vec{U}$ and $\vec{V}$. In order to traverse the rays front to back, which allows early termination when all rays have intersected intersection before the next slice, all rays must have the same sign along the traversal direction. For coherent packets, this is not a limitation; to violate this assumption, two rays would need to span an angle of more than $\frac{\pi}{2}$. We do *not* demand that all rays in a packet have the same dominating axis, nor that their direction signs match along $\vec{U}$ or $\vec{V}$, as is usually required by kd-tree packet traversers [Wald 2004] as long as the rays are coherent.

Now, consider a slice k along the major traversal axis, $\vec{K}$. For each ray in the packet, there is a point p_i^{in} where it enters this slice, and a point p_i^{out} where it exits. The axis aligned box $\mathcal{B}$ that encloses these points will also enclose all the $3D$ points – and thus, the cells – visited by at least one of of the rays. Once $\mathcal{B}$ is known, truncating its minimum and maximum coordinates yields the u, v extents of all the cells on slice k that are overlapped by any of the rays (Figure 3d).

Extension to Frustum Traversal Instead of determining the overlap $\mathcal{B}$ based on the entry and exit points of *all* rays, we can compute the four planes bounding the packet on the top, bottom, and sides. This forms a bounding frustum that has the same overlap box $\mathcal{B}$ as that computed from the individual rays. Since the rays are already transformed to grid-space, we can determine our bounding planes based on the minima and maxima of all the rays' u and v slopes. For a packet of $N \times N$ primary rays, we can simply compute

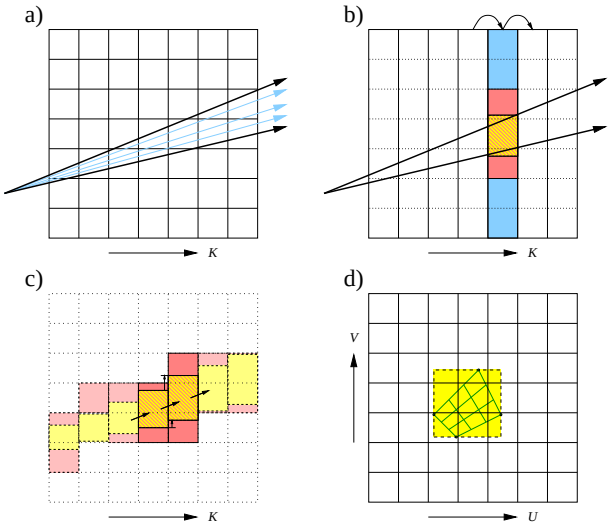

Figure 3: Given a set of coherent rays, our algorithm first computes the packet's bounding frustum (a) that is then traversed through the grid one slice at a time (b). For each slice (blue), we incrementally compute the frustum's overlap with the slice (yellow), which determines the actual cells (red) overlapped by the frustum. (c) Independent of packet size, each frustum traversal step requires only one four-float SIMD addition to incrementally compute the min and max coordinates of the frustum slice overlap, plus one SIMD float-to-int truncation to compute the overlapped grid cells. (d) Viewed down the major traversal axis, each ray packet (green) will have corner rays which define the frustum boundaries (dashed). At each slice, this frustum covers all of the cells covered by the rays.

these extremal planes using the four corner rays; however for more general packets all rays must be considered.

Traversal Setup Once the plane equations are known, we can intersect the frustum with the bounding box of the grid; the minimum and maximum coordinates of the overlap determine the first and last slice that should be traversed. If this interval is empty, the frustum misses the grid, and we can terminate without traversing.

Then, we compute the minimum and maximum u and v coordinates of the entry and exit points with the first slice to be computed. Essentially, these describe the lower left and upper right corner of an axis-aligned box bounding the frustum's overlap with the initial slice, $\mathcal{B}^{(0)}$. Note that we only need the u and v coordinates of each $\mathcal{B}^{(i)}$, as the k coordinates are equal to the slice number.

Incremental Traversal Since each slice's overlap box $\mathcal{B}^{(i)}$ is determined by the frustum's planes, the minimum and maximum coordinates of two successive boxes $\mathcal{B}^{(i)}$ and $\mathcal{B}^{(i+1)}$ will differ by a constant vector $\Delta\mathcal{B}$. With each slice being 1 unit wide, this $\Delta\mathcal{B}$ is simply $\Delta\mathcal{B} = (du_{min}, du_{max}, dv_{min}, dv_{max}$, where the $du_{min/max}$ and $dv_{min/max}$ are the slopes of the bounding planes in the grid coordinate space.

Given a slice's overlap box $B^{(i)}$, we can now incrementally compute the next slice's overlap box $B^{(i+1)}$ via $B^{(i+1)} = B^{(i)} + \Delta B$. This requires only four floating point additions, and can be performed with a single SIMD instruction]. As mentioned above, once a slice's overlap box B is known, the range $[i_0..i_1] \times [j_0..j_1]$ of overlapped cells can be determined by truncating $\mathcal{B}$'s coordinates and converting them to integer values. This operation can also be performed with a single SIMD float-to-int conversion instruction. Thus, for arbitrarily sized packets of $N \times N$ rays, the whole process of computing the next slice's overlapped cell coordinates costs only two instructions: a SIMD addition, and a SIMD float-to-int conversion. The complete algorithm is sketched in Figure 3.

2.3 Efficient Slice and Triangle Intersection

Once the cells overlapped by the frustum have been determined, we intersect all of the rays in a packet with the triangles in each cell. Triangles may appear in more than one cell, and some rays will traverse cells that would not have been traversed without packets. Consequently, redundant triangle intersection tests are performed. The overhead of these additional tests can be avoided using two well-known techniques: SIMD frustum culling, and mailboxing.

SIMD Frustum Culling A grid does not conform as tightly to the geometry than a kd-tree, and thus performs some triangle intersections that a kd-tree would avoid (see Figure 4). To allow for interactive grid builds, cells are filled if they contain the bounding boxes of triangles rather than the triangles themselves, further exacerbating this problem. (see Section 3). However, as once can see in Figure 4, many of these triangles will lie completely outside the frustum; had they intersected the frustum, the kd-tree would have performed an intersection test on them as well.

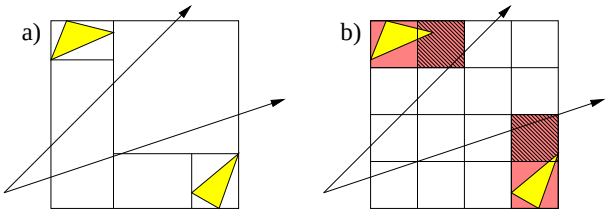

Figure 4: Since a grid (b) does not adapt as well to the scene geometry than a kd-tree (a), a grid will often intersect triangles (red) that a kd-tree would have avoided. These triangles however usually lie far outside the view frustum, and can be inexpensively discarded by inverse frustum culling during frustum-triangle intersection.

For a packet tracer, triangles outside the bounding frustum can be intersected quite cheaply using Dmitriev et al.'s "SIMD shaft culling" [Dmitriev et al. 2004]. If the four "corner rays" of the frustum miss the triangle on the *same* edge of the triangle, then all the rays must miss that triangle. Using the SIMD triangle intersection method outlined in [Wald 2004], intersecting the four corner rays costs roughly as much as a single SIMD 4-ray-triangle intersection test. As such, triangles outside the frustum can be intersected at $\frac{1}{N}$ the cost of triangles inside the frustum, where N is the number of rays in the packet.

Mailboxing In a grid, large triangles may overlap many cells. In addition, since a single-level grid cannot adapt to the position of a triangle, even small triangles often straddle cell boundaries. Thus, most triangles will be referenced in multiple cells. Since these references will be in neighboring cells, there is a high probability that our frustum will intersect the same triangle multiple times. In fact, as shown in Figure 5 this is much more likely for our frustum traversal than for a single-ray traversal: While a single ray would visit the same triangle only along one dimension, the frustum is several cells wide, and will re-visit the same triangle in all three dimensions.

Repeatedly intersecting the same triangle can be avoided by mailboxing [Kirk and Arvo 1991]. Each packet is assigned a unique ID, and a triangle is tagged with that ID before the intersection test. Thus, if a packet visits a triangle already tagged with its ID, it can skip intersection. Mailboxing typically produces minimal performance improvement in either a grid or a kd-tree for inexpensive primitive such as triangles; and may even reduce performance if

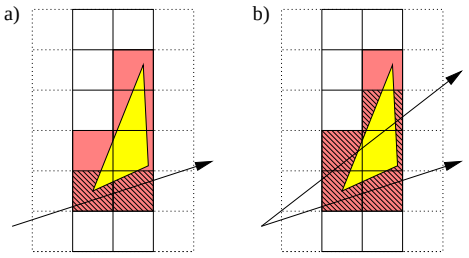

Figure 5: While one ray (a) can re-visit a triangle in multiple cells only along one dimension, a frustum (b) visits the same triangle much more often (even worse in 3D). These redundant intersection tests would be costly, but can easily be avoided by mailboxing.

gains from avoiding repeat intersection tests do not outweigh the costs of checking and updating the mailbox [Havran 2002].

As explained above, however, our frustum grid traversal yields far more redundant intersection tests than a single ray grid or kd-tree, and thus profits better from mailboxing. Additionally, the overhead of mailboxing for a packet traverser becomes insignificant; the mailbox test is performed *per packet* instead of per ray, amortizing the cost as we have seen before.

Impact of Mailboxing and Frustum Culling Mailboxing and frustum culling are both very useful in reducing the number of redundant intersection tests. In fact, both methods are much more powerful for our frustum grid traversal than for their original applications. Mailboxing is performed for multiple rays simultaneously, so the cost is amortized over the entire packet, and also avoids more redundant intersection tests. Similarly, due to the higher number of redundant triangle intersections in the packetized grid, SIMD frustum culling is more beneficial than in a kd-tree, where these intersections may have been avoided in the first place.

To quantify the magnitude of this impact, we have measured statistics on example scenes, using both a well-built kd-tree system employing 4×4 packets, and our frustum grid also using 4×4 packets. For each of those, we have measured the total number of ray-triangle intersections that are performed if neither of these techniques are used, then the results when mailboxing and finally SIMD frustum culling are applied. As can be seen from Table 1, mailboxing alone reduces the number of tests by up to a factor of 2; for a kd-tree, it usually trims this by less than 10% [Havran 2002]. On top of the reductions achieved by mailboxing, frustum culling achieves yet another reduction by a factor of 4 to 9. With both techniques, the final number of intersection tests decreases by a factor of 8.5 to 14, and the absolute number of ray-triangle intersection tests roughly matches that of a kd-tree.

Together, mailboxing and frustum traversal remedy the deficiencies of frustum traversal on uniform grids. Only one source of overhead cannot be avoided: when the bounding box of a triangle overlaps some cells traversed by a ray, but does not fall entirely outside the frustum. This scenario, however, is not limited to the grid; it also occurs in a packetized kd-tree.

scene	#tris	grid			ratio	kd-tree
MB/FC		-/-	+/-	+/+	-/- to +/+	+/+
ben	78K	12.8	6.0	1.51	8.5	1.09
hand	15K	12.5	6.0	0.93	13.4	0.85
toys	11K	14.0	8.7	1.0	14.0	0.82
conf	274K	96.0	53.9	6.9	13.9	3.66

Table 1: Ray-triangle intersection tests (in millions) for both a 4×4 kd-tree and for our 4×4 frustum-grid traversal, and the impact of mailboxing (MB) and frustum culling (FC). Mailboxing and frustum culling reduce the number of ray-triangle intersections by up to a factor of 14, to roughly as few as performed by a good kd-tree.

2.4 Extension to Hierarchical Grids

This algorithm is described for a singe-level grid; however hierarchical grids generally achieve superior performance. There are several ways to organize grids hierarchically, including loosely nested grids [Cazals et al. 1995] [Klimaszewski and Sederberg 1997], recursive or multiresolution grids [Jevans and Wyvill 1989], and macrocells or multigrids [Parker et al. 1999a]. Though these terms are ill-defined and often used ambiguously, they all share the same idea of subdividing some regions of space more finely than others, and thus traverse empty space more quickly than populated space. To demonstrate that our approach is not restricted to uniform grids, we have extended it with a single-level macrocell layer. Macrocells are a simple hierarchical optimization to a base uniform grid, often used to apply grids to scalar volume fields [Parker et al. 1999a]. Macrocells superimpose a second, coarser grid over the original fine grid, such that each macrocell corresponds to an $M \times M \times M$ block of original grid cells. Each macrocell stores a boolean flag specifying whether any of its corresponding grid cells are occupied.

Building the macrocell grid is trivial and virtually cost-free. Traversing it with our algorithm is rather simple: the macrocell grid in essence is just an $M \times M \times M$ downscaled version of the original grid, and many of the values computed in the frustum setup can be re-used, or computed by dividing by M. During traversal, we first consider a slice of macrocells, and determine all the macrocells overlapped by the frustum. (In practice, the frustum usually overlaps one macrocell). If the macrocells in our slice are all empty, we can skip M traversal steps on our original fine grid. Otherwise, we perform these steps as usual.

Though the best value of M obviously depends on the scene, $M = 6$ has consistently shown to be a good choice for the test scenes in our system. For smaller resolutions, the savings for each macrocell step become too small to justify the additional computations; for larger resolutions the probability of finding empty regions decreases. Using macrocells yields a performance improvement of around 30%, which is consistent with improvents seen for single ray grids. Additional levels of macrocells and could improve performance for more complex models with larger grids. More robust varieties of hierarchical grid could speed up large scenes with varying geometric density, at the cost of higher build time. As our goal is to formulate a viable grid traversal for medium-size animated scenes, these have not yet been investigated.

3 Acceleration Structure Rebuild

With an animated scene, our acceleration structure is recreated every frame. Though schemes for incrementally [Reinhard et al. 2000] or hierarchically [Lext and Akenine-Möller 2001] updating a grid exist, we did not want to impose any restrictions on the kind of animations we support, and thus opted for the most general method by rebuilding the grid from scratch for every frame. We use the common scheme of choosing the number of cells to be a multiple, λ, of the number of triangles, N [Devillers 1988; Cleary et al. 1983]. Due to having the smallest surface area in relation to volume, cubically shaped cells minimize a grid's expected ray tracing cost. Thus, we choose the grid's resolution as:

$$N_x = d_x \sqrt[3]{\frac{\lambda N}{V}}, N_y = d_y \sqrt[3]{\frac{\lambda N}{V}}, N_z = d_z \sqrt[3]{\frac{\lambda N}{V}},$$

where $\vec{d}$ is the diagonal and V the volume of our grid. Fortunately, our experiments show that most scenes are insensitive to the parameter λ and achieved their best performance around $\lambda = 6$ (Figure 6), which we use for all the experiments throughout this paper.

Once the grid resolution is chosen, for each triangle we determine the cells overlapped by the triangle's bounding box and add

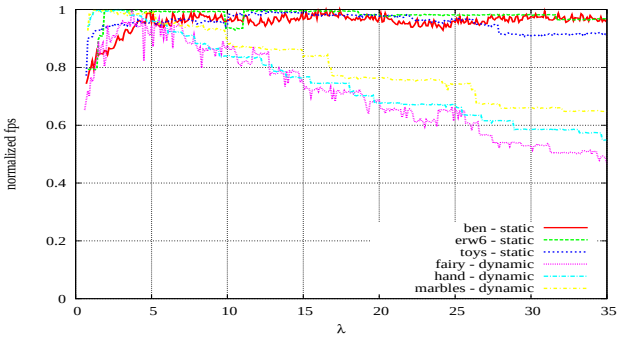

Figure 6: For several different models, this graph shows the framerate, normalized by the best time, in relation to grid size as determined by λ. Nearly all tested scenes, both static and dynamic, reach their optimum at approximately $\lambda \approx 6$.

a reference to the triangle to each of these cells. Since this is quite conservative, we also tested a more exact grid insertion scheme using an exact triangle-in-box test (e.g., [Akenine-Möller 2001]). However, though the exact test could reduce the number of triangle references in the grid by more than one third, the number of ray-triangle intersection tests – after mailboxing – would only shrink by a few percent. For such a small reduction in ray-triangle tests, the significantly higher rebuild cost does not pay off, leading us to use the less accurate – but faster – bounding box test.

Since memory allocations are costly, we use a preallocated pooled-memory scheme that prevents per-cell memory allocations and fragmentation as the scene changes from frame to frame. Memory layout techniques such as bricking [Parker et al. 1999b] have also been tested; but since the frustum traversal already amortizes memory accesses over the entire packet, these techniques did not result in a measurable performance difference for our scenes. Larger grids, however, may still benefit from these techniques.

In addition to rebuilding the grid, we also need to create the derived data for the triangle test described in [Wald 2004]. Though this could be avoided by storage-free triangle tests [Möller and Trumbore 1997], we found these to be slightly inferior in performance even when the per-frame triangle rebuild time is taken into account. Furthermore, the triangle rebuild takes less time than the grid rebuild, and can be run in parallel with the grid rebuild on a multi-CPU system.

4 Experiments and Results

In addition to the statistics presented above, we evaluated the performance of our algorithm on a working implementation. We first discuss the impact of the different governing parameters, and present the performance of the system for a range of both static and dynamic scenes. If not mentioned otherwise, all these experiments are performed at 1024×1024 pixels, and on a dual 3.2 GHz Intel Xeon PC with 3 Gigabytes of RAM.

4.1 Impact of Grid and Packet Resolution

For any given scene, the performance of our frustum traversal algorithm is governed by four factors: The resolution of the grid, macrocell resolution, screen resolution, and ray packet size. As shown in the previous section, choosing the grid resolution via $\lambda = 6$ in practice works fine for all kinds of scenes. Similarly, extensive experiments show that a macrocell resolution of $6 \times 6 \times 6$ yields reasonable performance for all tested scenes. Though tweaking these parameters can sometimes result in small performance gains, these default parameters usually work well.

While grid and macrocell resolution do have an impact, screen resolution and packet size have the greatest impact on performance. For any given packet size, the cost of a traversal step is constant, but the cost for intersecting the cells in a slice increases with the number of cells that the frustum overlaps. Larger packets will benefit more from the constant cost traversal step, but are also more likely to overlap more cells. Thus, there is a natural crossover point where the savings in traversal steps from a larger packet are offset by the additional cell intersections. Obviously, this crossover point will be influenced by the model resolution, as larger models have finer grids and correspondingly smaller cells.

To find that crossover point – and thus determine the optimal packet size – we generated different resolutions of the Stanford Armadillo model, and measured the rendering performance for packet sizes of 2×2, 4×4, 8×8, 16×16 and 32×32 rays per packet. The results of these experiment are given in Figure 7. For 2×2 packets the benefit of tracing packets is rather small, and the rendering times correspondingly high. Also not surprisingly, for the packets of 32×32 rays the packets get very wide, and performance deteriorates quickly as model complexity increases. Packets of 16×16 rays are better, but still deteriorate quite quickly. Somewhat surprisingly, both 4×4 and 8×8 perform similarly well. Though there is a crossover point around 250,000 triangles where the smaller packets work slightly better, the difference is not dramatic, and both 4×4 and 8×8 work well for the test scenes.

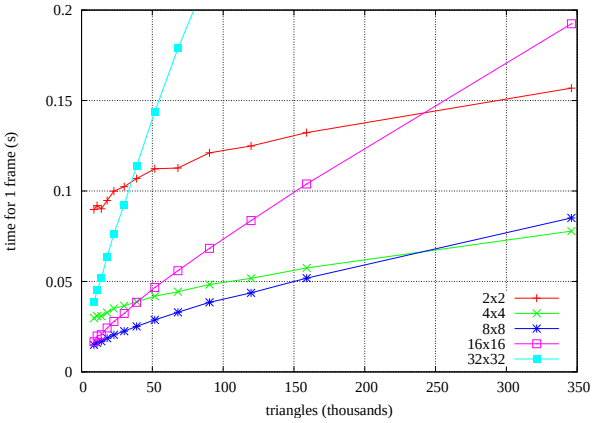

Figure 7: This graph shows the static scene render time per frame for different packet sizes, and for different resolutions of the Stanford Armadillo. As can be seen, there is a crossover point around 250K triangles where 4×4 packets become more efficient than 8×8 packets. Nevertheless, both 4×4 and 8×8 show nearly the same performance over a wide range of model complexity.

4.2 Scalability with Screen Resolution

Obviously, the optimal packet size also depends on the screen resolution, as higher resolutions result in a higher density of rays, and thus allow for larger packet sizes. Given today's hardware constraints, we chose 1024×1024 pixels as a default resolution for all our experiments. In the future, high-resolution displays and supersampling will push demand for even larger images.

While the cost of ray tracing is usually considered to be linear in the number of pixels, this is not the case for our algorithm. Since higher resolutions enable larger packets, we generally see sublinear scaling in screen resolution. For virtually all of our test scenes, when increasing the screen resolution from 1024×1024 to 2048×2048 the frame rate drops by only a factor of 1.75-2.25, significantly less than the expected factor of 4. Weakening the linear dependence on pixel count helps overcome a major hurdle in interactive ray tracing systems.

4.3 Performance for Static Scenes

Though our main motivation was to enable ray tracing of dynamic scenes, the performance gains achieved by the packet traversal apply also to static models. To evaluate our raw ray tracing performance, we used several typical static test models for ray tracing, and rendered them with our system with the rebuild disabled. This lets us consider traversal time independently from grid build time, and facilitates a comparison between our algorithm and contemporary interactive ray tracing systems, namely OpenRT [Wald 2004] and Intel's MLRT system [Reshetov et al. 2005].

For this comparison, we chose the erw6, conference, and soda hall scenes of 800, 280K, and 2.2M triangles, respectively, as these are the only scenes for which numbers from both systems are available [Reshetov et al. 2005]. Though the axis-aligned features of these three architectural models strongly favor the kd-trees used in MLRT and OpenRT, Table 2 shows that our system, despite relatively little low-level optimization, is competitive even for these best-case scenarios for the other systems.

scene	#tris	OpenRT 1 Pentium IV 2.5 GHz	MLRT 1 Pentium IV 3.2 GHz w/ HT.	Frustum Grid 1 Pentium IV 3.2 GHz w/ HT
erw6	804	2.3	50.7	18.3
conf	274k	1.93	15.6	4.0
soda hall	2.2m	1.8	24.1	7.4

Table 2: Static scene ray tracing performance for both the packetized grid, OpenRT, and MLRT. OpenRT and MLRT Data are taken from [Reshetov et al. 2005]; all times are including simple shading, but without display. Though these three scenes are best-case examples for our competitors, we remain at least competitive.

4.4 Comparison to Single-Ray Grid Traversal

The surprising performance of our frustum grid on architectural models can be explained by the benefits of packetization. To illustrate this difference, we compare our approach to an optimized single-ray 3DDDA implementation of a hierarchical grid. Though this implementation uses a more sophisticated multilevel hierarchy, Table 3 shows that the packetized grid ranges from 4 to 21 times faster, depending on the scene and viewpoint. Though some of this improvement is due to our use of SIMD extensions that cannot easily be used with single-ray traversal, SIMD implementation alone usually gives only about a factor of two; the remainder is due to cost amortizations and the algorithmic improvements of the packet/frustum technique.

scene	ben	hand	toys	erw6	conf
single-ray	1.57	1.59	1.53	0.670	0.302
8×8 packets	10.6	16.1	20.0	14.0	3.2
ratio	6.75	10.1	13.1	20.9	10.6

Table 3: Static scene performance (in frames per second) for our system; and for an optimized 3DDDA single-ray grid, using a macrocell hierarchy if advantageous. Images rendered at 1024×1024 pixels on a Pentium IV 3.2 GHz CPU with 1 thread and simple shading. Our frustum traversal outperforms the single-ray variant by up to an order of magnitude.

This can best be explained by the number of cells visited during traversal: as we see in Table 4, compared to a single ray traversal, the frustum version visits roughly 10 to 20 times fewer cells for the 4×4 packets, and over 50 times fewer for the 8×8 packets. Due to efficient packetized slice and triangle intersection (Section 2.3), the frustum actually tests fewer triangle intersections as well; and can even do that in SIMD.

scene	ben	hand	toys	erw6	conf
# ray-triangle intersection tests (millions)					
single ray	2.96	3.58	1.97	8.9	15.7
packet 4×4	1.50	0.93	1.02	1.54	6.90
packet 8×8	5.74	2.54	2.23	2.00	20.70
# visited cells (millions)					
single ray	24.3	19.6	7.72	33.2	167.7
packet 4×4	2.91	0.95	0.80	2.18	16.54
packet 8×8	1.37	0.36	0.32	0.58	5.84
ratio 4×4	13.1	20.74	9.65	15.23	10.13
ratio 8×8	8.35	54.9	23.9	55.7	28.7

Table 4: Total number of triangles intersected and cells visited (in millions) for a single ray grid; a 4×4; and an 8×8 packet traversal. No macrocells are being used by either grid, and tests use identical dimensions for the same scene. Frustum traversal dramatically reduces both the numbers of cell visits and triangle intersection tests.

4.5 Performance for Animated Scenes

To support animation, the simplest mechanism for a grid is to rebuild the grid structure every time the geometry changes. For small to medium sized scenes, rebuilding the grid is fast; allowing the performance achieved for static scenes to be sustained during animation. For larger scenes, other techniques such as incremental or parallel rebuilds may be required to maintain interactive performance, although these techniques were not employed here. To demonstrate these performance characteristics, we used several animated scenes of various sizes and different dynamic behavior, and measured the rebuild time and rendering performance. In the following, all performance data corresponds to 1024^2 pixels, with a simple shader as proposed in [Reshetov et al. 2005], but without shadows or display.

Animated meshes Some of the benchmark scenes are depicted in Figure 8: The "wood-doll" is a simple model with 5,378 triangles, and can easily be rendered at 30 frames per second. However, consisting only of rigid body animation of its otherwise static limbs, the wood-doll could also be rendered using rigid-body animation schemes for kd-trees as proposed in [Wald et al. 2003].

To stress more complex kinds of animation, we also tested an animated "hand" model of 15K triangles, as well as a "ben", a runner character of 80K triangles. Though already non trivial in size, the grid for the"ben" model can be rebuilt in 39ms, and around 13-15 frames per second are achieved during rendering, depending on the actual pose and viewing parameters.

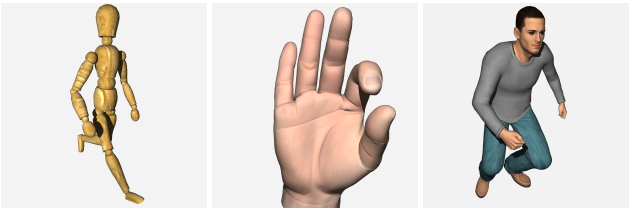

Figure 8: Some of the simpler animated models: a rigid-body wood-doll (5,378 triangles), a gesturing hand (15,855 triangles), and a running poser figure (78,029 triangles). Without shading and shadows, these scenes render at 56.7, 29.3, and 13.1 frames per second (including grid rebuild), and still at 30.1, 14.2, and 7.7 frames per second with shading, texturing, and shadows turned on.

Animated scenes Though differing in their forms of animation, both "wood-doll", "hand", and "ben" are individual models that are tightly enclosed by the grid. To demonstrate that our method is not limited to such models, the "toys" scene has a set of 5 individually animated wind-up toys that walk around incoherently, bump into each other, and even jump over each other (see Figure 9). With a total of 11K triangles, grid rebuild took 5ms, and a framerate of 20-34 frames per second is achieved.

The grid's strongest advantage over other data structures for handling dynamic models is that it does not require any kind of a hierarchy to be present in the model. Thus, it can also be used for completely incoherent motion of triangles, such as explosions, physics-driven simulations, or particle sets. To demonstrate this, we modelled a scene where 110 "marbles" are dropped into a (invisible) glass box, where they participate in a rigid-body simulation (Figure 9). Since the grid does not depend on any kind of coherence in the motion, this kind of animation can be supported easily, and 35-47 frames per second are achieved.

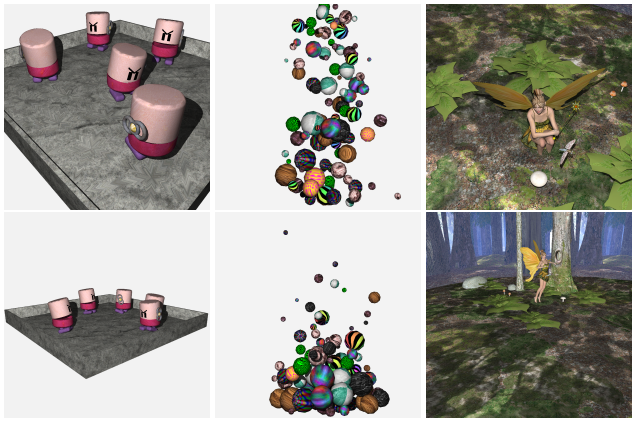

Figure 9: Examples of complex scenes composed of multiple individual objects: a) wind-up toys walking around and colliding with each other (11.1K tri); b) A simulation of 110 marbles dropping into an (invisible) box (8.8K tri). c) A complex scene of a typical game scenario: A fairy and a dragonfly dance through an (animated) forest; both fairy and dragonfly are animated based on a skinned skeleton (197K tri total). Without shading, these animations run at 20.0-34.1, 34.6-47.2, and 3.1-4.3 fps, respectively, and still at 8.1-16.2, 18.3-25.2, and 1.6-2.2 frames per second if shading, texturing, and shadows are turned on.

A real-world example While all these scenes are more or less artificial test models, the "fairy forest" scene (see Figure 9) has been chosen in particular because of its similarity to typical interactive scenarios: In this scene, a fairy and a dragonfly dance through an animated forest; both fairy and dragonfly are animated via a skinned skeleton. The scene incorporates both locally dense and largely empty regions; it is rather wide in spatial extent, requires complex shading, and consists of a total of 180K triangles, most of which are animated. Initially, we expected the high variation in scene density to be quite a challenge for our approach. However, the frustum traversal did surprisingly well, and still achieved some 3-4 frames per second, even under full animation.

The scenes discussed above were all modeled offline as animation sequences. This fact is not exploited at all by our traverser. The grid itself is built from a list of triangles and vertex positions every frame, neither knowing nor caring where they originate. It does not exploit the the temporal coherence properties of sequenced animation, but also does not depend on it. Thus, the system would work just as well for completely dynamic models. The number of triangles in the scene can easily be changed from frame to frame, and there is no restriction on the movement of existing triangles.

4.6 Impact of Shading and Shadows

In all the results so far, we considered only primary rays. However, shadow rays can easily be supported. For most rendering algorithms, coherent shadow rays can be generated by connecting all of the primary rays' hit points to the same point light source [Wald et al. 2001]. These packets then share a common origin just like primary rays, and differ from primary rays only in that they have no concept of "corner rays". However, we can easily use the four corners of the frustum to perform our SIMD frustum culling. In effect, shadow packets and primary rays packets behave exactly the same.

Obviously, the performance for shadow rays depends significantly on the coherence in the packet: If the primary ray packet has strafed the silhouette of a foreground object, the 3D hitpoints may be quite distant, and connecting those to the same point light may result in quite wide and incoherent packets. As proposed in [Wald et al. 2001], such packets can be detected by looking at the primary rays' minimum and maximum hit distances; the packets can then be split if this difference exceeds a certain threshold. For many of the scenes that we tested, this was not necessary since the reduced coherence affected a small fraction of the packets. However, some lighting configurations slowed our system substantially when shadows were applied, indicating that a splitting technique would be required to restore the coherence in the packet.

More general packets that do not even share the same origin would be possible, as long as the rays are still coherent. Though this would imply the method also works at least for perfect reflections or highly glossy materials, this has not yet been tested.

5 Summary and Discussion

We presented a new approach to ray tracing with uniform grids. This algorithm elegantly allows for transferring the recent advantages in fast ray tracing – namely, ray packets, frustum testing and SIMD extensions – to grids, for which these techniques had previously not been available. The frustum based grid traversal has several important advantages. First, it has a simple traversal step, where a few SIMD operations allow for determining all the cells in a grid slice that are overlapped by the frustum. This operation has a constant cost for the entire frustum that is amortized over the entire packet of rays, and allows for a traversal step that is at least as cheap as that of a packet/frustum kd-tree. Using mailboxing and SIMD frustum culling (Section 2.3), our method performs roughly the same number of ray-triangle intersection tests as the kd-tree. Though our implementation is not *as* highly tuned as that of Intel's MLRT system [Reshetov et al. 2005], it is up to 21 times faster than known single-ray grid traversal schemes; competitive with kd-trees; and inherently supports fully-dynamic animated scenes.

Our method does possess several limitations. The very nature of using a uniform grid makes the method ill-suited for highly complex scenes with a high variation in size and density of geometry – e.g., the Boeing data set [Gobbetti and Marton 2005] or the classic teapot-in-a-stadium. Though our macrocell technique works for most cases, for highly complex scenes multiresolution grids [Parker et al. 1998], multilevel techniques [Wald 2004; Lext and Akenine-Möller 2001], or separation of static and dynamic objects [Reinhard et al. 2000], as well as mechanisms to incrementally rebuild the grid data structure may be advantageous.

Grids still suffer from common pathological cases such as large flat areas (i.e., from architectural models) where geometry overlaps numerous cells. These situations can be handled more efficiently by today's kd-tree based ray tracers and therefore, kd-trees are likely to remain somewhat more efficient for many scenes. It is also unclear how the proposed technique will perform for secondary rays (i.e. reflection and refraction), where the coherence is lower than in primary and shadow rays.

Our technique may be very appropriate for special-purpose hardware architectures such as GPUs and the IBM Cell processor [Minor et al. 2005] that offer several times the computational power of our current hardware platform. Though kd-trees have been realized on both architectures [Foley and Sugerman 2005], are limited by the streaming programming model in those architectures. In contrast, a grid-based iteration scheme is a better match to these architectures, and may be able to achieve a higher fraction of their peak performance. The current method may be appropriate for a hardware-based implementation, similar to [Woop et al. 2005]. The grid has a reduced dependence on stack memory that may reduce the resources required in a hardware implementation.

The primary motivation of this approach is to enable ray tracing of dynamically deforming models. Rebuilding an acceleration structure on each frame enables ray tracing these models without placing constraints on the nature and range of the motion. As this update cost is – like rasterization – linear in the number of triangles, it introduces a natural limit for the size of models that can be rebuilt interactively. The rebuild cost is manageable for many applications such as visual simulation or games, where objects of several thousand to a few hundred thousand polygons are common. The ability to support these kind of models makes interactive ray tracing practical for an entirely new class of applications.

Acknowledgments

The 'fairy forest' animation has been created with DAZ Studio; the base models have been graciously provided by DAZ Productions (www.daz3d.com). The 'toys' and 'marbles' animations are modeled by Andrew Kensler. All scenes are available via the Utah Animation Repository (http://www.sci.utah.edu/~wald/animrep). We would like to thank Alexander Reshetov for insight into his MLRT system.

Miscellaneous

An accompanying demonstration video is available online at http://www.sci.utah.edu/~wald/SuppVideo_grid.mov.

This paper has been conditionally accepted at ACM SIGGRAPH 2006. Once available, the revised version will replace this technical report. Please check back for the revised version.

References

AKENINE-MÖLLER, T. 2001. Fast 3D triangle-box overlap testing. *J. Graph. Tools 6* (1), 29–33.

AMANATIDES, J., AND WOO, A. 1987. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics '87*. Eurographics Association, 3–10.

CAZALS, F., DRETTAKIS, G., AND PUECH, C. 1995. Filtering, Clustering and Hierarchy Construction: a new solution for Ray Tracing very Complex Environments. *Computer Graphics Forum (Proceedings of Eurographics '95) 14* (3).

CLEARY, J., WYVILL, B., BIRTWISTLE, G., AND VATTI, R. 1983. A Parallel Ray Tracing Computer. In *Proceedings of the Association of Simula Users Conference*, 77–80.

DEVILLERS, O. 1988. *Méthodes d'Optimisation du Tracé de Rayons*. PhD thesis, Université de Paris-sud.

DMITRIEV, K., HAVRAN, V., AND SEIDEL, H.-P. 2004. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

FOLEY, T., AND SUGERMAN, J. 2005. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of HWWS*, 15–22.

FUJIMOTO, A., TANAKA, T., AND IWATA, K. 1986. ARTS: Accelerated ray tracing system. *IEEE CG&A 6* (4), 16–26.

GLASSNER, A. S. 1984. Space subdivision for fast ray tracing. *IEEE CG&A 4* (10), 15–22.

GLASSNER, A. 1989. *An Introduction to Ray Tracing*. Morgan Kaufmann. ISBN 0-12286-160-4.

GOBBETTI, E., AND MARTON, F. 2005. Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 24* (3), 878–885.

HAVRAN, V. 2001. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.

HAVRAN, V. 2002. Questions and Answers: Mailboxing, Yea or Nay? *Ray Tracing News 15* (1).

JEVANS, D., AND WYVILL, B. 1989. Adaptive voxel subdivision for ray tracing. *Proceedings of Graphics Interface '89* (June), 164–172.

KIRK, D., AND ARVO, J. 1991. Improved ray tagging for voxel-based ray tracing. In *Graphics Gems II*, J. Arvo, Ed. Academic Press, 264–266.

KLIMASZEWSKI, K. S., AND SEDERBERG, T. W. 1997. Faster ray tracing using adaptive grids. *IEEE CG&A 17* (1) (Jan./Feb.), 42–51.

LEXT, J., AND AKENINE-MÖLLER, T. 2001. Towards rapid reconstruction for animated ray tracing. In *Proceedings of Eurographics*, 311–318.

MINOR, B., FOSSUM, G., AND TO, V. 2005. TRE : Cell broadband optimized real-time ray-caster. In *Proceedings of GPSx*.

MÖLLER, T., AND TRUMBORE, B. 1997. Fast, minimum storage ray triangle intersection. *JGT 2* (1), 21–28.

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. 1998. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98*, 233–238.

PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P.-P., HANSEN, C., AND SHIRLEY, P. 1999. Interactive ray tracing for volume visualization. *IEEE Transactions on Computer Graphics and Visualization 5* (3), 238–250.

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.

PURCELL, T., BUCK, I., MARK, W., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *Proceedings of SIGGRAPH*, 703–712.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering*, 299–306.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *(Proceedings of SIGGRAPH*, 1176–1185.

SPACKMAN, J., AND WILLIS, P. 1991. The SMART navigation of a ray through an oct-tree. *Computers and Graphics 15* (2), 185–194.

SPACKMAN, J. 1990. *Scene Decompositions for Accelerated Ray Tracing*. PhD thesis, The University of Bath, UK. Available as Bath Computer Science Technical Report 90/33.

STOLL, G. 2005. Part II: Achieving Real Time - Optimization Techniques. In *SIGGRAPH 2005 Course on Interactive Ray Tracing*, P. Slusallek, P. Shirley, I. Wald, G. Stoll, and B. Mark, Eds.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. In *Proceedings of Eurographics*, 153–164.

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

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

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