Incoherent Ray Tracing without Acceleration Structures

Attila T. Áfra[1,2]

[1]Budapest University of Technology and Economics, Hungary
[2]Babeş-Bolyai University, Cluj-Napoca, Romania

Abstract

Recently, a new family of dynamic ray tracing algorithms, called divide-and-conquer ray tracing, has been introduced. This approach partitions the primitives on-the-fly during ray traversal, which eliminates the need for an acceleration structure. We present a new ray traversal method based on this principle, which efficiently handles incoherent rays, and takes advantage of the SSE and AVX instruction sets of the CPU. Our algorithm offers notable performance improvements over similar existing solutions, and it is competitive with powerful static ray tracers.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

A ray tracer typically consists of two main parts: ray traversal and acceleration structure building. Rendering fully dynamic scenes with ray tracing is a particularly challenging problem since the acceleration structure must be either updated or rebuilt for every frame. There has been extensive research on this topic [WMG*09].

Keller and Wächter [KW11] recently proposed a largely different and elegant approach: *divide-and-conquer (DAC) ray tracing*. Its basic idea is to incorporate the primitive partitioning into the ray traversal algorithm. Thus, no acceleration structure is required. Unfortunately, the authors did not provide many algorithmic details or performance results. The only published full implementation to date is by Mora [Mor11]. It partitions the primitives in a kd-tree-like manner, but does not use the high-quality surface area heuristic (SAH) [Hav00].

In this paper, we propose a new DAC traversal algorithm based on the core method by Keller *et al.* Our approach is generally more efficient than Mora's method, and it exploits the power of the new AVX (Advanced Vector Extensions) instruction set introduced with the Intel Sandy Bridge microarchitecture. We have optimized our method for incoherent rays because most rays shot by a Monte Carlo renderer (e.g., path tracer, photon mapper) have little or no coherence. This not only improves the performance of physically based rendering, but also simplifies the overall design.

2. Divide-and-Conquer Ray Traversal

This ray traversal algorithm is practical only if a large number (ideally millions) of rays are traced together, in breadth-first fashion. However, a notable advantage of DAC traversal is that the number of partitioning steps can be significantly lower than for the building of an acceleration structure. While it is necessary to build the entire tree regardless of the actual ray distribution, DAC traversal performs partitioning only for regions that are visited by rays.

The inputs of the algorithm are the set of rays and the set of primitives to intersect. At the beginning of each traversal step, we determine which rays intersect the bounding box of the primitives. This operation is called *ray filtering*, and the rays that intersect the box are *active rays*. Next, we decide whether to further partition the primitives or to directly compute their intersections with the active rays. If we have subdivided the primitives, we recursively call the traversal function for the resulting subsets.

3. Our Algorithm

A key part of the traversal algorithm is primitive partitioning. Kd-trees are constructed using *spatial partitioning*, while BVHs mostly use *object partitioning*. An important feature of object partitioning is that it divides a list of primitives into disjoint sublists, contrary to spatial partitioning. BVHs are more suitable for dynamic scenes than kd-trees because they can be built up to an order of magnitude faster, and consume

DOI: 10.2312/conf/EG2012/short/097-100

less memory since they are shallower [Wal07]. Therefore, we opted for object partitioning in our traversal algorithm.

We use triangles as primitives in our implementation, but the method can be easily adapted to other types of primitives.

3.1. Ray filtering

The purpose of ray filtering is to determine the list of active rays, which intersect a given axis-aligned bounding box (AABB). This list initially contains all rays, and it is recursively filtered in each traversal step. The filtering can be executed in-place by rearranging the list to create an active and an inactive partition. This way, the active rays start at the beginning of the list, and an offset to the end of the list is enough to identify them.

We keep the rays in a single continuous array. Since our method has to trace millions of rays at the same time, the rays occupy a considerable amount of memory. This affects not only the space requirements, but also the speed.

A ray is specified using a point of origin, a direction vector, an interval $[0, t_{\max}]$ defining a line segment, and an ID. The total size of a ray is 32 bytes, which means that it fits into a single AVX register or two SSE registers. Most ray tracing algorithms precompute the inverse ray direction, but for our method, this is not practical, and the increased ray size degrades performance.

Mora [Mor11] filters rays by reordering an auxiliary ID array to avoid moving the much larger ray structures. Although this approach is preferable for coherent rays and small arrays, it does not work well for arrays of incoherent rays larger than the cache. The number of cache misses, due to random memory accesses, can be significantly reduced by using software prefetching, but cache space is still wasted. The CPU prefetches entire cache lines (usually 64 bytes), which are larger than a single ray. If not all rays in the cache line are needed, half of the prefetched data is wasted.

We avoid these caching problems by simply reordering the rays in the original array. Much more data is moved than in the previous technique, but no extra information is needed, and the cache is utilized highly effectively. As the rays are always accessed linearly, hardware prefetching, which is triggered automatically, is sufficient. Rays can be quickly copied in blocks of 32 (with AVX) or 16 bytes (with SSE). This filtering approach is best suited for incoherent rays, in which case it improves ray traversal performance by at least 25%.

The reordering makes it impossible to identify rays with their array indices, which is necessary for accessing additional information linked to the rays (e.g., intersection data). Fortunately, this does not pose a problem as there is enough room in the ray data structure to store an ID.

SIMD instruction sets enable faster filtering by carrying out multiple ray/box intersections at the same time. In our implementation, we simultaneously intersect 4 rays when using SSE and 8 rays when using AVX. Before doing so, the ray data, which consists of 8 values per ray, must be rearranged into structure-of-arrays (SoA) format. We again leverage the SIMD-friendly layout of the rays to quickly transpose the values with SIMD shuffle instructions.

The AABB that is tested against the rays is the bounding volume of the current list of triangles. Computing this box right before filtering is simple to implement, but it introduces an additional sweep over the triangle list. We avoid this by computing the AABB during the triangle partitioning step.

3.2. Triangle partitioning

Triangle partitioning divides a list of triangles into two disjoint sublists with the goal of minimizing the total number of intersections. We use two different partitioning methods: *middle partitioning* and *SAH partitioning*. Both split the list of triangles according to the centroids of the triangle AABBs. The splitting axis is always the one in which the centroids' AABB is widest.

The partitioning algorithms do not process the triangles themselves, but only their AABBs. To avoid computing the AABB of a triangle several times throughout the traversal, and to optimize the cache usage, we precompute the AABBs and put them into a separate array. We speed up the computations with the AABBs by storing them in a SIMD-friendly 32-byte format: the minimum and maximum bounds (which are 3D vectors) are padded to 16 bytes so they can be directly loaded into SSE registers.

In contrast with ray filtering, we manage a triangle ID array instead of directly reordering the AABBs. Multiple renderer threads operate on the same primitives, but each needs a separate array to work with. Hence, we store the AABBs only once and use multiple ID arrays, one per thread.

A simple and very fast way to partition the triangles is middle partitioning. This consists of splitting the centroid AABB at its spatial median and sorting the triangles into the resulting partitions. While sweeping over the array, we also precompute for each partition the AABB of the triangles and the AABB of the box centroids.

For some splits, we use SAH partitioning instead, which usually results in lower intersection costs, but it is slower. We employ the SAH binning algorithm [Wal07] with 32 bins, which is much faster than the approximation-free approach [Hav00, WBS07] and produces almost as good results.

Always partitioning with the SAH does not necessarily lead to the highest possible ray tracing performance because its computational cost may be too high compared to the cost of the ray intersections. We solve this problem by adaptively deciding between SAH and middle partitioning. In each partitioning step, the ratio of the number of active rays and current triangles is checked against a predefined threshold. If this value is above the threshold, SAH partitioning is chosen.

Otherwise, middle partitioning is used. We have empirically found out that a ray/triangle ratio between 1 and 2 delivers the best results.

3.3. Triangle intersection

If a certain stopping criterion is met, the triangles are not partitioned further, but are naively intersected with the active rays. We stop if either the active ray count or the current triangle count is too low. A threshold of 8 is used for both counts. The SAH can also stop the partitioning.

In our method, a special triangle representation is used to save memory space and bandwidth. A useful property of triangle AABBs is that they are defined with 6 vertex coordinates of the triangles: the minimum and maximum vertex coordinates for each axis. In an additional array we store the missing data for each triangle. This complementary data structure contains the 3 remaining vertex coordinates and a 32-bit integer encoding the original order of the coordinates needed for the reconstruction of the triangle vertices.

For each ray, the traversal algorithm has to compute and store the closest intersection with the geometry. To effectively use the cache, we update $t_{\max}$ in the original ray structure with the intersection distance. The remaining intersection data is located in an array separate from the rays because it is less frequently accessed.

The intersection routine is composed of two nested loops. The outer loop iterates over the list of triangles, and the inner loops iterates over the list of active rays. The cost of triangle decoding is negligible as it is amortized over several rays. Similarly to the ray filtering routine, multiple rays are intersected with the current triangle using SIMD.

3.4. Ordered traversal

After partitioning the triangle list, the traversal state for one of the sublists is pushed onto a stack, and the traversal continues with the other one. The traversal state includes the AABB of the triangles, the AABB of the box centroids, the triangle range, and the active ray range.

For primary rays, front-to-back traversal has a significant positive impact on the ray tracing speed. However, the improvement is small for incoherent rays. We determine the traversal order with the very cheap approach from the packet tracer by Wald *et al.* [WBS07]. First, we find the axis on which the two child boxes are furthest apart. Then, we get the first active ray's direction sign on that axis, and select the first child along the respective direction.

4. Results

The benchmarks were run on two different systems: on an Intel Core i7-960 (*Nehalem*, 4 cores, 8 threads, 3.2 GHz, 8 MB L3 cache) with 24 GB RAM (triple channel), and on an Intel Core i7-2600 (*Sandy Bridge*, 4 cores, 8 threads, 3.4 GHz, 8 MB L3 cache) with 8 GB RAM (dual channel).

We compared our method to a highly optimized static ray tracer that uses the multi-BVH (MBVH) structure [WBB08]. We built the MBVHs using the same partitioning schemes as in our method. For up to 16K triangles, we used binned SAH splitting. Otherwise, middle splitting was employed.

We tested the algorithms using a 1-bounce and an 8-bounce Monte Carlo path tracer with diffuse reflections. (The number of bounces indicates the maximum ray recursion depth.) The length of the rays was infinite, and the closest intersections were found. Russian roulette was not used.

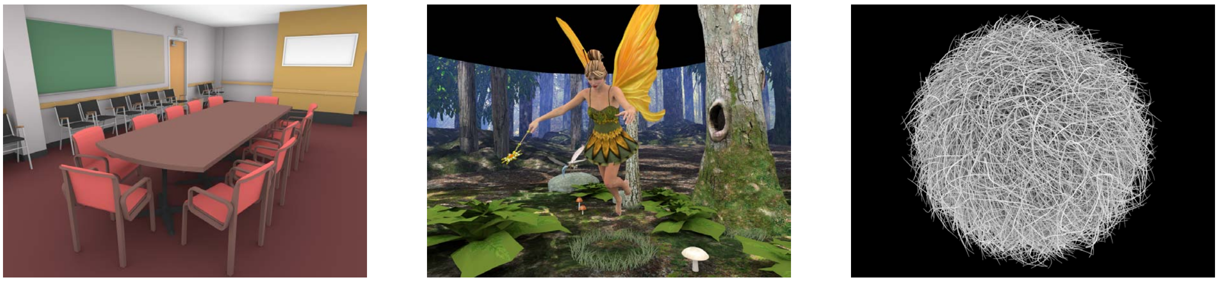
Our multithreaded renderer processes one path per pixel for the entire image in each thread. This naive multithreading approach is suboptimal for DAC traversal because a portion of the triangle partitioning steps are executed multiple times on different threads. However, this is not an issue if there are enough rays to amortize the cost of triangle partitioning. A more efficient solution has not yet been proposed.

The performance results, in million rays per second, are listed in Table 1. The timings for the MBVH method do *not* include the acceleration structure build time. Even so, our method is still quite competitive in most cases. For example, MBVH is only 12% faster for the 8-bounce path tracing of the CONFERENCE ROOM scene, on a single thread of the i7-960. However, the difference is greater on multiple threads, especially for HAIRBALL, where our method is 4× slower.

Another important observation that can be made is that AVX provides a notable speedup over SSE. Previous research showed that it improves ray tracing speed for primary rays by about 50% [Áfr11]. For diffuse rays, the AVX version of our method is 0–47% faster than the SSE version. The speedup for MBVH traversal is also good: 15–31%.

The results also show that traditional ray traversal scales better to multiple threads than DAC traversal. While MBVH scales *superlinearly* with the number of cores, our method scales *sublinearly* in some cases. Surprisingly, the i7-2600 mostly scales worse than the older i7-960, possibly because of its slightly inferior memory system. For one scene, both the SSE and AVX versions on the i7-2600 are even slower than the SSE one on the i7-960. The multithreading speedup is heavily scene-dependent: on i7-960 it is 3.4–4.9×, and on i7-2600 it is 1.8–5.3×. The sublinear scaling suggests that memory bandwidth is the primary performance bottleneck.

Ray filtering is typically the most time-consuming part of our algorithm. For CONFERENCE ROOM, ray tracing is composed of 70% ray filtering, 19% triangle intersection, and 11% triangle partitioning. In such cases, naive multithreading is quite efficient because partitioning has a small impact on the overall performance. However, for complex scenes like HAIRBALL, partitioning is more prominent: 37% filtering, 25% intersection, and 38% partitioning. (Note that there are almost 4× more triangles than rays.)



(a) CONFERENCE ROOM (282K triangles) (b) FAIRY FOREST (174K triangles) (c) HAIRBALL (2880K triangles)

Figure 1: *Scenes used for the performance measurements.*

Method	SIMD	CPU	CONFERENCE ROOM				FAIRY FOREST				HAIRBALL			
			1-bounce		8-bounce		1-bounce		8-bounce		1-bounce		8-bounce	
			ST	MT	ST	MT	ST	MT	ST	MT	ST	MT	ST	MT
Our	SSE	Core i7-960	1.7	5.8	1.7	6.2	1.4	6.3	1.3	6.4	0.3	1.4	0.2	0.8
MBVH4	SSE	Core i7-960	2.2	12.2	1.9	11.4	1.9	10.9	1.8	10.8	0.7	4.1	0.5	3.2
Our	SSE	Core i7-2600	2.0	5.2	2.0	5.8	1.7	6.1	1.6	6.7	0.3	1.6	0.2	0.8
Our	AVX	Core i7-2600	2.9	5.3	2.8	5.8	2.5	6.5	2.2	7.5	0.4	1.8	0.2	1.0
MBVH4	SSE	Core i7-2600	2.9	16.1	2.6	14.9	2.5	14.0	2.5	13.9	0.9	5.1	0.7	4.0
MBVH8	AVX	Core i7-2600	3.8	20.4	3.3	18.5	3.1	17.1	3.0	16.6	1.1	6.1	0.9	4.6

Table 1: *Performance in million rays per second (Mray/s) for diffuse rays generated with 1-bounce and 8-bounce path tracing (no Russian roulette). The scenes were rendered from the views shown in Figure 1 at* 1024×768 *resolution. Both single-threaded (ST) and multithreaded (MT) speeds were measured. The timings do not include ray generation, shading, and MBVH building.*

Compared to Mora's method, our approach filters rays more efficiently, uses higher quality object partitioning, exploits wider SIMD, and is optimized for incoherent rays. On one thread, on slightly higher clocked CPUs, for CONFERENCE ROOM, our SSE code is 1.3–1.5× faster, while the AVX one is 2.2× faster. The methods use similar amounts of memory. For R rays, T triangles, and N threads, our method uses $32R+(48+4N)T$ bytes, whereas Mora's uses $36R+(40+4N)T$ bytes (excluding the stack and hit data).

5. Conclusions and Future Work

We have presented a new divide-and-conquer ray tracing algorithm optimized for incoherent rays, which does not use any acceleration structures. Our method outperforms the only other similar solution and is competitive against standard algorithms that require major precomputations.

The main drawback of DAC ray tracing methods is that they are more difficult to efficiently parallelize than traditional approaches, and thus they do not work optimally with *naive* multithreading. In the future, we would like to develop a better multithreaded variant of the algorithm.

Acknowledgements

This work has been supported by POSDRU/107/1.5/S/76841 and by OTKA K-719922 (Hungary).

References

[Áfr11] ÁFRA A. T.: Improving BVH ray tracing speed using the AVX instruction set. In *EG 2011 - Posters* (Llandudno, UK, 2011), Eurographics Association, pp. 27–28. 3

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. 1, 2

[KW11] KELLER A., WÄCHTER C.: Efficient ray tracing without auxiliary acceleration data structure. *High-Performance Graphics 2011 (Poster)* (2011). 1

[Mor11] MORA B.: Naive ray-tracing: A divide-and-conquer approach. *ACM Transactions on Graphics 30*, 5 (October 2011), 117:1–117:12. 1, 2

[Wal07] WALD I.: On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 33–40. 2

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets – Efficient SIMD single-ray traversal using multibranching BVHs. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing* (2008), pp. 49–57. 3

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (January 2007), 6:1–6:18. 2, 3

[WMG*09] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the art in ray tracing animated scenes. *Computer Graphics Forum 28*, 6 (2009), 1691–1722. 1