Naive Ray-Tracing: A Divide-And-Conquer Approach

BENJAMIN MORA
Swansea University

We present an efficient ray-tracing algorithm which, for the first time, does not store any data structures when performing spatial subdivisions, and directly computes intersections inside the scene. This new algorithm is often faster than comparable ray-tracing methods at rendering dynamic scenes, and has a similar level of performance when compared to static ray-tracers. Memory management is made minimal and deterministic, which simplifies ray-tracing engineering, as spatial subdivision data structures are no longer considered in the graphics pipeline. This is possible with a modification of Whitted's naive ray-tracing algorithm by using a divide-and-conquer approach, and by having a sufficient collection of rays in order to reduce the complexity of naive ray-tracing. In particular, the algorithm excels at spontaneously solving large Ray/Primitive intersection problems.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Animation*

General Terms: Algorithms

Additional Key Words and Phrases: Ray tracing, divide-and-conquer, rendering, global illumination

ACM Reference Format:

Mora, B. 2011. Naive ray-tracing: A divide-and-conquer approach. ACM Trans. Graph. 30, 5, Article 117 (October 2011), 12 pages.
DOI = 10.1145/2019627.2019636
http://doi.acm.org/10.1145/2019627.2019636

1. INTRODUCTION

Ray-Tracing (RT) is a fundamental and much investigated computer graphics algorithm for solving primitive/ray intersections. Unlike rasterization, RT can accommodate any ray parameterization, which makes it useful to numerous rendering problems. The basic naive algorithm to solve a RT problem is well-known and consists of intersecting each ray with all the primitives inside the scene. However, the trivial complexity of this algorithm (O(*primitives* × *rays*)) is not suitable for most applications that nowadays handle large quantities of rays and triangles. As such, naïve RT has been quickly discarded and spatial subdivision data structures have been proposed and widely investigated to reduce the per-ray complexity.

Nevertheless, the compulsory usage of an additional data structure in RT can be seen sometimes as a burden by developers, and makes RT software engineering more complex. Therefore, a new method based on the naïve algorithm that can directly solve intersections between a set of rays and a set of primitives at no additional memory cost is of much interest, especially if the method is proven fast. To make this possible, an observation was made that a complexity of O(*primitives*) to find a ray's closest intersection (naïve algorithm) does not necessarily imply a complexity of O(*primitives* × *rays*) for a problem including several rays where a new divide-and-conquer approach can actually reduce the solution space. Indeed, it can be shown that a naive RT function can be broken down into several noticeably smaller naïve RT functions (Figure 1(b)), hence a recursive divide-and-conquer definition for the base case. This new approach for naïve RT has led to an algorithm that represents a significant contribution to RT in the following areas.

—*New RT Paradigm*. This article presents a new and very simple RT technique not using a spatial subdivision data structure for the first time since Whitted [1980]. Its divide-and-conquer principles can potentially be applied to all spatial subdivision schemes studied to date. Also, RT is achieved in a breadth-first quicksort-like streaming fashion, contrasting with normal tracing techniques which mainly resort to depth-first recursive traversals.

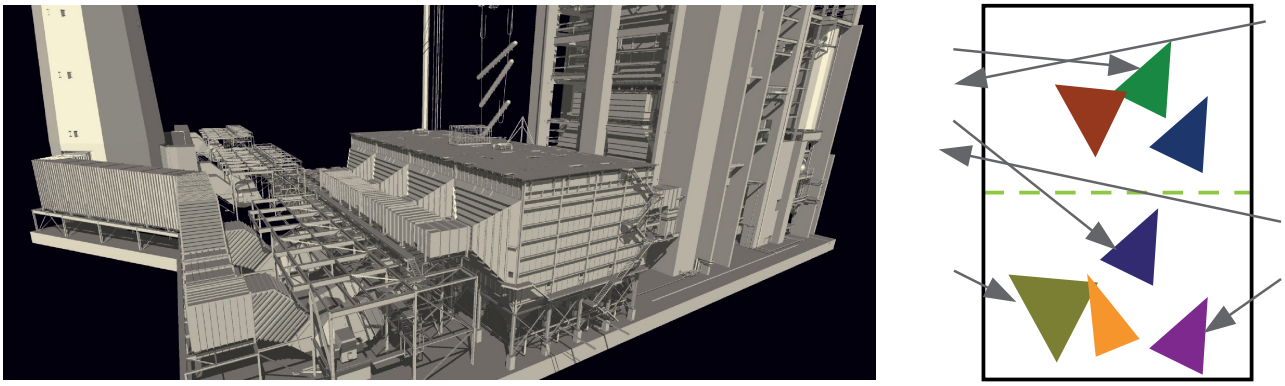
—*Rendering Speed*. The breadth-first double streaming approach combined with new optimizations provides an inherently efficient algorithm that can be up to an order of magnitude faster for dynamic scenes and is almost on-par with static ray-tracers on equivalent hardware.

—*Minimal and Deterministic Memory Usage*. The memory requirement of our new ray/primitive intersection algorithm is minimal and can be determined in advance as a linear function of the number of primitives and the number of rays. This permits solving of larger problems and can be of interest especially when specific rendering hardware with limited on-board memory is involved. Flexibility of ray-tracing is improved because a preprocessing step is not required and intersections are directly computed, hence simplifying the software engineering side of RT, especially in dynamic scene contexts.

It must be noted that while our Divide-And-Conquer Ray Tracing (DACRT) algorithm does not require a spatial subdivision data structure to be stored, it still involves spatial subdivisions as the divide-and-conquer scheme. As such, we will formalize in Section 3 the principles of the DACRT algorithm and describe a first implementation based on Axis-Aligned Spatial Subdivisions (AASS). The term AASS is used in preference to the term kd-trees throughout this article to emphasize that we do not refer to a stored data structure. New optimizations to RT will also be presented and results provided in Section 4 will then show that DACRT associated with AASS is very fast for rendering dynamic scenes.

B. Mora acknowledges EPSRC grants EP/E001750/1 and EP/I005870/1.
Author's address: B. Mora, Swansea University, Singleton Park, SA2 8PP, Swansea, UK; email: b.mora@swan.ac.uk.

DOI 10.1145/2019627.2019636
http://doi.acm.org/10.1145/2019627.2019636



(a) (b)

Fig. 1. Divide-and-conquer ray-tracing quickly solves large and dynamic intersection problems without using a spatial subdivision data structure. (a) The 12.7 million triangles of the power plant model are rendered on a 6.6 millions pixel image (3840 × 1728) with a single 3 GHz core in 1.4s with primary rays only, and 3.4s when a single-point light source is activated. (b) A ray-tracing solution computed naively will require 42 intersection tests (7 × 6). If space is cut in half, we can advantageously replace this naïve solution with two new naïve solutions only requiring 28 intersection tests combined (4 × 3 + 4 × 4). Repeating such a scheme recursively as long as the solution space is too large allows direct and fast computing of intersections.

2. PREVIOUS WORK

With the exception of the first real RT implementation [Whitted 1980], all RT algorithms that have followed Whitted implementation have been closely associated with spatial subdivision data structures to significantly lower complexity. Early proposed spatial subdivisions included kd-trees [Bentley 1975], Bounding Volume Hierarchies (BVH) [Rubin and Whitted 1980; Kay and Kajiya 1986], BSP-trees [Fuchs et al. 1980], Octrees [Glassner 1984], and grids [Fujimoto et al. 1986]. In contrast, DACRT makes no use of such data structures and therefore cannot be directly compared to previous techniques. Despite this, there are many recent works that aim at similar objectives, including rendering in a dynamic context, reducing memory usage, or providing more flexible software approaches.

For rendering static scenes, RT has usually been considered a very efficient rendering technique if the construction times of the acceleration data structure are ignored. However, recent efforts have focused more on the rendering of dynamic scenes, and therefore construction times must now be considered. This raises a few issues as long construction times are likely to provide a better tracing rate, while faster construction times may see tracing times degrading. As such, Wald et al. [2006a] recognized that a good balance could be difficult to find, and proposed the use of grids as an original alternative to the long build times of hierarchical space subdivision data structures such as kd-trees. As grids are renowned for a lower tracing complexity of $O(n^{1/3})$ [Cleary and Wyvill 1988; Ize et al. 2007], a fast packet traversal was also proposed. Further work on grids has then been produced by Lagae and Dutré [2008], focusing more on both the grid construction times and low memory usage. Results produced in this article clearly show that grid structures can be computed at an extreme speed and that rendering times are more ray-bounded. However, some discrepancy in rendering times given by Lagae and Dutré [2008] is visible, and grids seem to perform noticeably better for scenes with isotropic triangles, as predicted by Ize et al [2007]. To remedy these "pathological cases" [Wald et al. 2006a], structures like kd-trees may be preferred.

Kd-trees are slower to construct than grids, but Hunt et al. [2006] and Shevtsov et al. [2007] have introduced a method of getting an interactive construction that even includes a very fast SAH estimation. The tracing part of the Shevtsov et al. [2007] algorithm was managed by the very fast MLRT algorithm [Reshetov et al. 2005], with the observation that a reduction in the tree quality could be suitable if fewer rays are to be traced. Zhou et al. [2008] have overcome the task of porting the kd-tree construction on GPUs, showing some clear acceleration. In comparison with grids, the use of kd-trees seems much more primitive-bounded, as a lot more rendering time is generally spent constructing the spatial subdivision data structure and much less in tracing. For instance, Garanzha and Loop [2010] managed to trace soft shadow rays at a rate up to 150M rays/s.

For animated scenes, the reconstruction of a full data structure at each frame may be inefficient, and a simple update of the data structure is sometimes more logical. Wald et al. [2006a] and Lauterback et al. [2006] thus independently demonstrated that bounding volume hierarchies are well suited to this task, although the time needed to compute the original BVH is not interactive for moderately large scenes. Hunt and Mark [2008a;2008b] more recently demonstrated that a frustum grid subdivision can also be highly efficient.

A drawback to all previously mentioned approaches is that the memory footprint may vary between frames, which may be a disadvantage with regard to software engineering, as dynamic allocation must then be involved. Instead, many approaches do not include the cost of dynamic allocation in their discussion and just make use of preallocated memory. Nonetheless, Lagae and Dutré [2008] presented a framework for dynamic reallocation with limited impact. The importance of memory management was acknowledged by Wächter and Keller [2006; 2007] who proposed Bounding Interval Hierarchies (BIHs) where the maximum memory usage is linearly bounded by the number of triangles. Although the memory usage is not exactly deterministic, this BIH property is possible because there is no need to replicate triangle indices. An original, on-demand construction was also demonstrated, where only a partial tree is constructed before tracing. During the backtracking tracing process, the construction is eventually completed for temporary leaf nodes that are reached, possibly avoiding processing occluded regions of the scene. As a result, the memory needed for the BIH is typically only a fraction of that needed for the scene. Wächter and Keller [2007] also showed how kd-tree construction criteria can be modified to

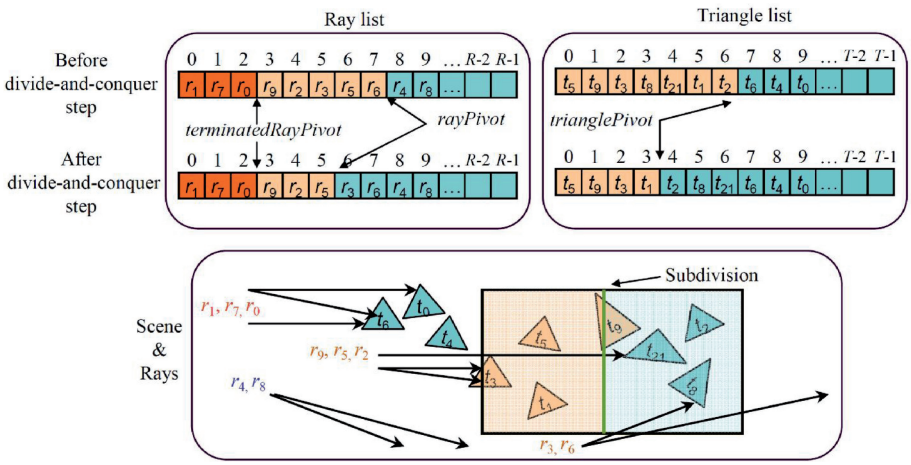


Fig. 2. In-place computations showing one divide-and-conquer step that filters/sorts rays and triangles in the left subdivision. Note that the *rayPivot* variable will need to be recomputed for the right subdivision in the second pass later on and is stored on the stack, while the *terminatedRayPivot* variable is global.

allow spatial subdivisions to fit into a constrained memory block. They also mentioned not using an acceleration data structure in a patent Wächter and Keller [2009]. Our work has been developed independently, without prior knowledge of this patent.

3. PRINCIPLES OF DIVIDE-AND-CONQUER RAY-TRACING

3.1 Formal Algorithm

The naïve RT algorithm consists of two nested loops computing every possible ray-triangle intersection in the scene at a complexity of O(*primitives* × *rays*). As this approach is processing far too many intersections, our idea is to simplify the problem by only intersecting subsets of rays with subsets of triangles, which will be determined by a divide-and-conquer scheme using spatial subdivisions as in Algorithm 1.

ALGORITHM 1: Divide-And-Conquer Ray-Tracing

```
procedure DACRT (Space E, SetOfRays R, SetOfPrimitives P)
begin
    if R.size() < rLimit or P.size() < pLimit
    then NaiveRT (R, T);
    else begin
        {E_i} = SubdivideSpace (E)
        for each E_i do
            SetOfRays R' = R ∩ E_i;
            SetOfPrimitives P' = P ∩ E_i;
            DACRT(E_i, R', P');
        end do
    end
end
```

The DACRT algorithm first compares the number of primitives (triangles here) and the number of rays involved in the problem with two arbitrary fixed constants. If one of the two comparisons is true, the algorithm just uses the naïve algorithm. Otherwise, space is subdivided and a recursive call is made for each subspace only including the primitives and rays intersecting the given subspace. Note that space subdivision is not restricted to an Euclidian 3D space, but could be performed for instance in either an image (2D) or light field (4D) space. Also, all computations for the intersection operators of Algorithm 1 can actually be performed in-place in a breadth-first [Nakamaru and Ohno 1997] quicksort-like fashion using pivots (Figure 3), with only a few additional Kilobytes needed for the recursion stack. If we leave the ray component out of the algorithm, we get the framework of a simple kd-tree construction algorithm, storage part not included.

3.2 Basic Implementation

For our purposes, we will suppose that the primitives are always triangles and that the associated space subdivision is a 3D AASS, as with kd-trees. AASSs were chosen because it is a common and well-proven technique with possibly some potential for DACRT, although the existence of superior subdivision schemes is likely. Other requirements for this implementation include in-place computations, front-to-back traversal associated with early-ray termination, and also specific optimizations like conic packet tracing (Section 3.3), fast triangle streaming (Section 3.4), and simplified split determination (3.5).

In-place computation is necessary as it renders the memory usage minimal and deterministic. If we except the recursion stack (which is a matter of a few kilobytes) the algorithm just requires triangles and rays to be linearly stored in two separated lists (Figure 2). Two pivots are needed at each recursion step to differentiate the triangles and rays that are either inside or outside the current subdivision. A third ray pivot is also needed for early ray termination. To reduce memory transfers by not moving triangle and ray data structures in main memory, two indexing arrays are also used. The memory usage is then increased at a cost of four additional bytes for each triangle or ray, but still remains deterministic.

At the heart of every new recursive call to the DACRT function is a filtering process that separately sorts triangles and rays in a breadth-first quicksort fashion (Figure 3). For triangles, indices are parsed between 0 and the old *trianglePivot*, and a new *trianglePivot* is then generated (stored on the stack) such that triangles indexed at the left-hand side of the pivot intersect the child subdivision investigated, while triangles at the right-hand side do not (Figure 2).

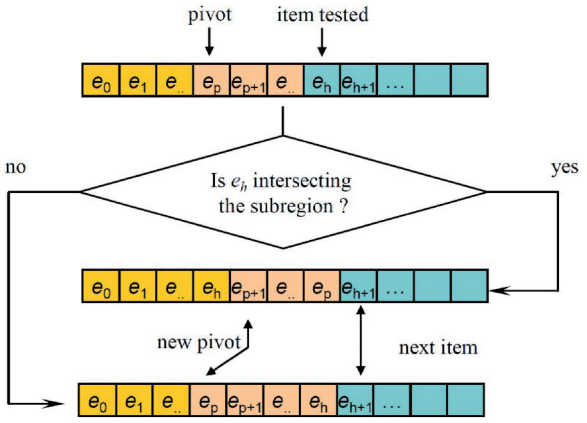


Fig. 3. In-place modification of a list (either a ray list or a primitive list) according to one intersection test between one element (either a ray, a cone, or a primitive) of the list and the subregion.

This filtering process must be executed for each subdivision of a given node, that is, twice in the current implementation.

The same filtering process is applied to rays, with the noticeable difference that only rays with indices between the *terminatedRayPivot* (global variable) and the old *rayPivot* are to be filtered. This stream filtering idea has been suggested by Gribble and Ramani [2008] for an SIMD context, though no real implementation to our knowledge has been made, and only one packet of rays (4^2 to 64^2) was handled. In contrast, our implementation streams the entire image/ray dataset. It must be noted, however, that, unlike regular depth-first traversals, our approach does not store tmin and tmax parameters to accelerate intersection tests as these two parameters must be stacked for each ray. As these would increase memory consumption and would make it nondeterministic, a complete ray/box intersection test is used instead.

Naïve ray-tracing is used whenever the problem size is small enough. The two constants *rLimit* and *pLimit* of Algorithm 1 have been respectively fixed to 20 and 16, as lower or higher values were found to degrade performance. The Möller and Trumbore [1997] intersection test is used to solve intersections, with a rate of approximately 25 clock cycles per intersection for our SSE version. Faster intersection implementations do exist, but typically increase memory requirements.

Early ray termination avoids computing intersections beyond the first successful intersection. To do this, spatial subdivisions must be investigated in a front-to-back order by processing the subdivision that is on the same side as the viewpoint first, and the other side next. When the naïve RT function is called, intersections between all the rays inside the *terminatedRayPivot* and *rayPivot* range (Figure 2), and all the triangles between the first index and the *trianglePivot* index are computed. If intersections are detected, rays are terminated by moving the *terminatedRayPivot* pivot to the right and adding the terminated ray indices to the left-hand side of the pivot (Figure 2). By using early ray termination, occluded triangles inside the scene are quickly discarded and their impact on the rendering times is significantly reduced.

Complexity of the basic algorithm (without packets) applied to axis-aligned subdivisions is bounded by the complexity of a simple kd-tree construction followed by a nonpacket RT algorithm. This comes from the observation that the algorithm will traverse the same spatial subdivision tree as a pure kd-tree construction algorithm, and for each triangle that is parsed in our algorithm, there would be an equivalent operation in the kd-tree construction algorithm. Note that the algorithm is bounded since invisible regions of the tree are not treated if no ray traverses them. Furthermore, each ray streaming operation (i.e., intersection test) corresponds to a single ray-node traversal in the regular ray-tracing algorithm. We will make no further analysis of the complexity as this is beyond the aim of this article, but results show that rendering problems where the naïve algorithm would normally perform hundreds of billions of intersections, requiring approximately half an hour of computations with our intersection test's implementation, can actually be solved in less than a second.

3.3 Conic Packets

A direct implementation of the algorithm quickly showed that the number of ray/box intersections is around 20 to 60 times the original number of rays, which makes rendering relatively slow for primary rays. In regular RT, pyramidal ray packets [Wald et al. 2001] are an important acceleration technique that significantly reduces the number of nodes traversed if both the renderer and the scene feature sufficient coherency (e.g., primary rays). However, running a packet/node intersection can be expensive, as the pyramid consists of several faces and this test is often simplified [Reshetov et al. 2005]. Also, determining the pyramid for secondary rays is a more complex task [Wald et al. 2006a]. We therefore decided to investigate conic packets, where the pyramid is replaced by a cone, allowing exact cone-box intersection tests. Some former uses of cones were investigated by Amanatides [1984] for antialiasing purposes and by Roger et al. [2007] for creating a ray hierarchy. Further research by Tsakok [2008] has demonstrated that, while slightly less efficient for primary rays than pyramidal packets, cones can be more efficient for secondary rays. Therefore, we have chosen to include conic packets in our pipeline. A simplified algorithm similar to Algorithm 1 and supporting conic packets is given in Algorithm 2. Note that the same method is used for computing shadow rays to a point light source, which simply requires redefinition of the cones from the light source and intersection points of each packet.

Cones are internally defined with a data structure that includes the main cone direction, an angle α, and a 64-bit variable that stores the termination status of each ray inside the packet. α is determined such that there exists no ray in the packet whose angle with the main cone direction is greater than α. The inclusion of cones inside our DACRT pipeline is easily achieved by replacing rays in Algorithm 1 by cones in Algorithm 2 and storing a new indexing list for cones. The packet size for testing purposes has been fixed to 8×8 rays, and a cone test may therefore replace at most 64 ray tests. For instance, where the original code would compute 1024^2 ray-box intersections for the root spatial subdivision, the new conic version will only compute 128^2 cone-box intersections.

The cone list initialization takes place immediately following the ray generation and prior to calling DACRT_Packet (Algorithm 2), with no modification afterward of the data structure. Cone termination bits are also set to zero, and the ray index list is considered empty at this point, contrasting with the original approach where ray indices initially had to be enumerated in a fully populated list. The cone list replacing the ray list is, however, fully enumerated at the beginning, and future ray indices that are generated with *FlushCones* will be deduced implicitly from the cone indices. The variables *terminatedRayPivot* and *terminatedConePivot* must be initialized to zero before recursion as well. At all times, indices in

the range [0.. *terminatedRayPivot*] will be representative of valid and final ray intersections.

During recursion, DACRT_Packet (Algorithm 2) will call DACRT (Algorithm 1) instead of directly calling *naiveRT* when a termination criterion is valid as this is more efficient for dense meshes. With regard to low-level implementation, the call to DACRT needs to be preceded and followed by two operations to take advantage of intersections detected early in the process.

First, it is necessary to copy the (nonterminated) ray indices related to each cone into the ray index list (after the *terminatedRayPivot* pivot) prior to the call to DACRT, which also determines the new value for *rayPivot*. Second, we need to update the cone termination bits for the positive ray-scene intersections found. Removal of all cones with all bits set to 1 after the DACRT call must also be performed by moving terminated cone indices to the left of the *terminatedConePivot* pivot. Positive intersections can be found from the ray indices between the successive positions of *terminatedRayPivot* before and after the call to DACRT, which allows setting termination bits to 1.

Intersections of cones with node/boxes is a simple but crucial step that must be performed efficiently during the streaming process. The technique first involves detecting whether the main cone direction intersects the box. If not, the algorithm must also investigate whether or not there is an intersection with any of the 12 edges of the box. If not, the cone does not intersect the box. At first glance, this process seems inefficient, however, it takes advantage of two factors. Firstly, the initial test is statistically often positive, avoiding the second step. Secondly, the 12 tests can be reduced to 0, 4, and 6 tests for cases where the viewpoint location is respectively inside the node, inside the node for 2 of the 3 spatial dimensions, or not matching the previous two cases. Such edges can be defined as the ones representing the contours of the projection of the node on the unit sphere centered on the viewpoint. Implementation-wise, the edge configuration is described in a table (64 entries, of which 27 are valid), and the active entry is determined with 6 tests comparing the viewpoint 3D coordinates with the 6 planes defining the bounding box of the node. The mathematics determining an edge-cone intersection [Eberly 2000] are given in the Appendix section, and involve solving a second-degree polynomial. Our current SSE implementation has an approximate rate of 80M intersections/s on a 3 GHz processor.

ALGORITHM 2: Simple Conic-packet version of DACRT

```
procedure DACRT_Packet (
            Space E, SetOfCones C, SetOfPrimitives P)
begin
    if C.size() < cLimit or P.size() < pLimit
    then begin
        SetOfRays R = FlushCones(C);
        DACRT (E, R, P);
    end
    else begin
        {E_i} = SubdivideSpace (E)
        for each E_i do
            SetOfCones C' = C ∩ E_i;
            SetOfPrimitives P' = P ∩ E_i;
            DACRT_Packet (E_i, C', P');
        end do
    end
end
```

3.4 Handling Secondary and Random Rays

The conic packet implementation relies on rays emitted from common locations in space (e.g., Frustum rays, punctual or virtual light sources). This allows for simplification of intersection tests and reduces bandwidth. Many algorithms, however, trace rays that are randomly located in space. To handle this scenario, we use a different, generic implementation of the algorithm, simplified in several ways.

First, conic packets are not used as there is no longer any real coherency. Rays are then defined from a starting point and a direction, which requires 6 floating-point values plus 2 more for memory alignments. Early ray termination is also deactivated and the *terminatedRayPivot* variable is not used as nodes will not necessarily be investigated in visibility order. Instead, we rely on a z-buffer-like algorithm. The t value that describes the intersection location for each ray is updated for each successful intersection test, and successful ray-box intersections that occur beyond the current t value are handled as failing intersections. Not using early ray termination also means that there is no traversal order anymore in the scene and hence, determining which of the left or right child will be investigated first is determined according to the number of axis-positive or axis-negative rays, as proposed by Reshetov [2006]. Another approach could have been to split rays into 8 directional subsets and still use front-to-back traversal for the 8 subsets, but the current approach has been satisfactory.

It must be noted that rays will still skip hidden areas, but in a more random manner. Indeed, rays will not be intersected with objects belonging to subdivisions behind an already detected intersection. However, the closest intersection may not be the first detected by the algorithm.

DACRT is efficient at handling large problems in one step and this may look incompatible with the recursiveness of some techniques. Optimal practice is to pack as many rays as possible into just one pass, which can be achieved with almost all ray-tracing techniques by cleverly reorganizing the workload. For instance, this can be done in a path-tracer by processing primary rays in one batch, then casting light-source rays in another batch, then casting all secondary rays from primary ray intersections in another batch, and so on. From a software engineering point of view, managing large chunks of rays can been seen as a trade-off with the management of a spatial subdivision data structure, though the batch processing nature of the workload here may be easier to handle.

3.5 Triangle Streaming and Parsing

Quickly streaming triangles is of high importance, especially if the scene is large. Our current implementation can stream approximately 300 million triangles per second and can perform as many simplified triangle-box intersections on a single 3 GHz core, which is enough to process multimillion triangle scenes interactively. For this task, we have chosen the triangle-box intersection method given by Akenine-Moller [2001], with many simplifications made. This test is based on the separating hyperplane theorem, with 13 specific plane orientations to be tested in our case. The triangle plane is one of them, but has been removed from the test as early results showed that it was a slow and not crucial test. Indeed, our storage scheme does not allow storing the triangle normals, which would then need to be computed on-the-fly for this purpose. This optimization does not affect correctness, though some useless triangles will be kept in the lists for longer than required.

Three other directions to be tested are given by the orientations of the bounding box's faces. As, again by induction, some of these

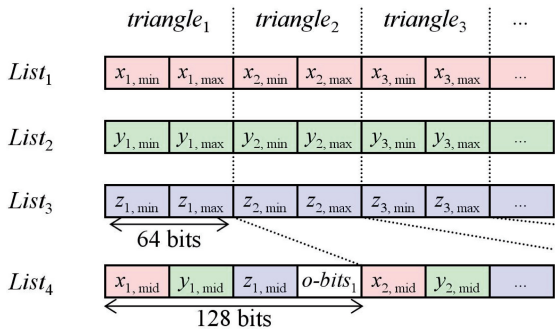


Fig. 4. Scene storage by decomposition into four memory-aligned lists.

intersections have been carried out at higher levels of the hierarchy, these tests can be replaced by a single test with the split plane.

The last 9 tests are based on directions given by the cross-products of the three triangle edges and the three cube edges. These nine tests are resource consuming, but we observed that for large streams, most triangles had their three vertices on the same side of the split plane. As such, these 9 tests are only carried out when the number of triangles in the stream is small enough. This number has been arbitrarily fixed at 100 and helps to reduce the number of forthcoming ray-triangle intersection tests considerably.

Therefore, only a single test with the splitting plane is needed in most streaming cases, which is extremely fast as four comparisons can be carried out in one SSE instruction. To speed up this test, a particular triangle storage has been designed. A standard 36-byte triangle representation has been chosen, with values reshuffled so that the triangle's min-max coordinates on each axis appear in separate lists (Figure 4). As such, only four 64-bit loads followed by one SSE shuffle operation are needed before testing four triangles at a time. Another four bytes are added in the fourth list for data alignment and also storing the bits required for reordering the vertex coordinates. Reordering is done on-the-fly whenever required (e.g., 12-case triangle-box intersections and triangle/ray intersections) and compression/decompression costs are negligible in comparison with other operations. Indeed, the slight time increase for reordering is largely compensated by the considerable decrease of the time needed for streaming triangles. Taking into consideration the 32-bit integer needed for indexing each triangle, the (deterministic) linear memory usage is 44 bytes per triangle. We have chosen this particular scheme as a good trade-off between memory consumption and rendering speed, though we could have either reduced or increased this requirement to the benefit of one of these.

3.6 Axis-Aligned Subdivision

The Surface Area Heuristic (SAH) is the best subdivision scheme known to date for kd-trees. Unfortunately, it is relatively slow at subdividing space, though it usually reduces the number of traversal steps during the tracing phase and is often worth the effort in regular RT. Instead, we favored brute-force streaming of triangles, with a simplified determination of the splitting axis and position. The axis is determined by the longest dimension of the bounding box. For the position itself, two different schemes are used according to whether less than 10000 triangles are to be scanned or not. For large streams, a 50^{th} of the triangles are evenly selected in the stream and are analyzed to estimate the split so that it adds only a small amount of time in comparison with the time needed for streaming

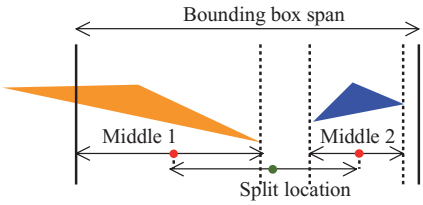


Fig. 5. Simple split location determination used by our algorithm.

and sorting all of the triangles. For streams that are too small (less than 10000 triangles), the estimation is not robust enough and the algorithm resorts to a simple space median cut. The analysis for large streams involves computing the middle of the triangle span (min-max on the selected axis) for each selected triangle, clamped by the bounding box limits first (Figure 5), and then computing the average position of all these points.

This approach is very simple, fast, and performs slightly better than a direct median cut (∼20%) with the constants mentioned before. We initially tried the technique proposed by Shevtsov et al. [2007], but an implementation of their algorithm in our context looked less attractive. We conjecture that the subdivision process is actually benefitting from the knowledge of already-instantiated rays, for instance, by stopping subdivision earlier when only a few rays are present. This contrasts with the SAH which considers uniform-only, nonquantifiable distributions.

4. RESULTS

Four different types of test have been carried out as they represent conditions where different levels of optimizations generally apply: primary rays only, single, point light source, and path-tracing and photon emission tests.

Most tests have been performed by an Intel-core 2 duo E6850 computer (3 GHz), equipped with 4 GBs of memory and an NVidia 8800 GTX graphics card with 768MBs. An intel core i7 (3 GHz) is used only for the multithreaded analysis. Our implementations (DACRT and the reference packet ray-tracer) are SSE3 optimized and monothreaded (single core used), unless otherwise specified.

Basic and display lists based OpenGL results are given mainly for scaling comparison, although we are aware that more efficient on-board storage may be possible. Rendering times for DACRT and Packet RT are given for the specific images shown and include basic RGB shading and shadow rays where appropriate. Note that for the same clock frequency, core 2 processors are significantly faster than P4 processors, even though the use of Hyper-Threading (HT) reduces the gap. Unless stated, all renderings use the conic packet acceleration.

4.1 Primary Rays and Single-Point Light Source

Our first test (Figure 6) reuses the methodology introduced by Wächter and Keller [2006], with the image size fixed to 640 × 480. Results show an important speedup for most models that actually tends to increase with problem size. DACRT can be up to 7.5× faster than an optimized kd-tree approach where construction and tracing are separated. The on-demand BIH approach is between 3× and 8× slower, but again, it is believed that using similar processors would reduce this gap.

In our second test (Figures 7 and 14), the image size is fixed to 1024 × 1024 and with more rays now involved, we expect a better efficiency. For space reasons, we could only compare a limited set of techniques in Figure 14 that tend to have similar hardware, but the

		Wächter & Keller P4HT	Shevtsov et al	DACRT Core 2
Erw6, 806 tri.		11 / 12 fps	Core 2 43 fps 0.91×	3 GHz 47 fps 1×
Bunny, 69K tri.		6 / 5.6 fps 0.3 / 0.28	9.6 fps 0.48 ×	19.6 fps
Dragon, 863K tri.		0.9 / 0.64 fps	1.3 fps 0.24 ×	5.4 fps 1×
Conference		1.5 / 0.65 fps	1.6 fps 0.13 ×	12 fps 1×
Buddha		1.4 / 0.54 fps	1.4 fps 0.14 ×	9.6 fps 1×

Fig. 6. Results for the Wächter and Keller [2006] tests including *fps* and relative performances (Ray-Casting, 640 × 480, Monothreaded). DACRT uses the conic packet optimization and BIH results are given for the *on demand* variant on/off.

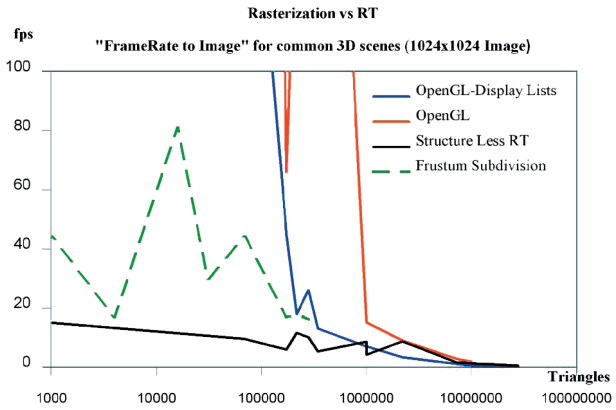


Fig. 7. Comparison of our RT algorithm for primary rays with Rasterization (NVidia GTX 8800 and results from [Hunt and Mark08], which use 1920× 1200 pixel window).

diversity of the measurements provided in this article should facilitate further comparisons with other approaches such as Havran et al. [2006] and Zhou et al. [2008]. For instance, Zhou et al. [2008] demonstrated similar levels of performance for relatively small scenes, but using an 8800 GTX ultra Graphics card benefiting from 50× more FLOPS and 10× more memory bandwidth than our test platform. It is nevertheless well known that, unlike CPU platforms, peak performance is rarely reached with GPUs, as programming them introduces many constraints.

As such, additional results are chosen from Shevtsov et al. [2007]; Lagae and Dutré [2008] for dynamic contexts, and Overbeck et al. [2008] for static MLRT [Reshetov et al. 2005] as results include one more dataset, and triangle or quad-based scene representations are tested. For Shevtsov et al. [2007], we included the kd-tree construction rate only, because different lighting conditions are used (e.g., fairy scene) which makes comparisons quite difficult.

Figure 14 shows that when compared to other ray-tracing dynamic methods (rasterization methods apart) our algorithm can be up to an order of magnitude faster, especially when large problems are to be solved. For instance, it takes 2.46s to compute just the kd-tree for the Thai statue with 4 cores in Shevtsov et al. [2007], while our algorithm needs less than 0.8s on a single core to achieve the final image. The difference is even bigger for the soda-hall scene; the tree is built at 2.1 fps on a four core system versus 8.7 fps for the full image on a single core. However, this difference may be at least halved if a shadow ray pass is needed. Logically, grids [Lagae and Dutré 2008] do much better for isotropic scenes (5s versus 2.7s only for DACRT with the Thai statue) due to a faster construction but tracing seems quite inefficient for scenes with nonisotropic triangles (e.g., cabin and conference scenes are respectively 30× and 21× slower). Finally, the idea proposed by Wald et al. [2006a] to update the data structure instead of reconstructing it from scratch seems to be very efficient, but constructing the spatial subdivision for the first frame requires much time and therefore it has a reduced domain of applications.

The comparison with static renderers, assuming an already-constructed spatial subdivision, tells us that the performance of our renderer is very good. The reference packet ray-tracer is on average slower than DACRT, although it can be up to 1.6× faster for the largest dataset. With an equivalent triangle-based scene representation, MLRT is apparently slower on average, but as an older-generation processor was used, MLRT is likely to be faster on a core 2 platform, though not by much. As such, it is clear that streaming rays and cones in a breadth-first way [Nakamaru and Ohno 1997] is a very powerful alternative to regular depth-first top-down traversals of spatial subdivisions. It is nonetheless worth noting that even in the case of large datasets, our approach is still competitive when compared to static renderers and can justify not using an acceleration data structure in order to benefit from easier memory management, which may, in fact, be of importance with these datasets.

A comparison with z-buffer techniques for primary rays (Figure 7, OpenGL and Hunt and Mark [2008a]) shows that rasterization is faster for small scenes. Indeed, Hunt and Mark's approach is between 2× and 4× faster for twice the number of pixels, but with backface culling enabled (halving the number of primitives) and using implicit ray coordinates. While performance is very high for small scenes, it is not shown that it can remain that competitive for larger and Soda-Hall-like scenes, where traditional grids do not usually perform well.

A rapid decrease of graphics hardware efficiency when the size of the scene increases can be observed in Figure 7, even though a single CPU cannot really compete with dedicated hardware for small scenes due to limited raw power. In the extreme case of the Lucy dataset (which cannot fit directly into the video memory) our algorithm is about two times faster than hardware-accelerated rasterization.

Figure 8 gives further details about the number of steps needed by our algorithm, and also includes rendering times with a point light source. For the basic algorithm, the respective number of triangle/box and ray/box intersections is typically an order of magnitude greater than the respective numbers of triangles and rays in each rendering problem. As such, the algorithm is logically ray-bounded when the number of rays is much higher than the number of triangles, and triangle-bounded in the opposite case. However, the introduction of conic packets decreases the number of ray and cone intersection tests by an order of magnitude. Of particular interest is the fact that rendering times approximately double when a

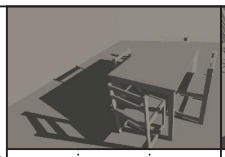
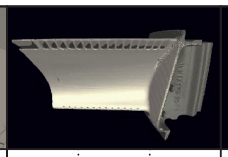
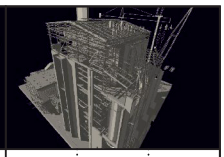
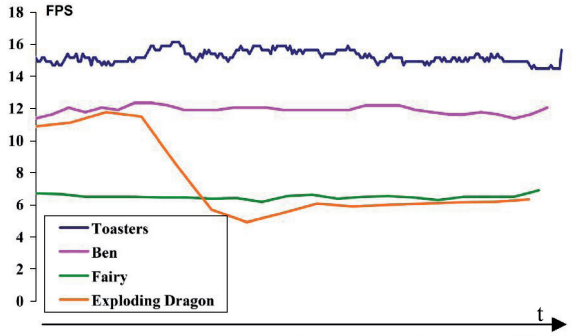
From left to right: **Erw6**, 806 tri. **Conf.**, 282K tri. **Blade**, 1.76M tri.												
	Basic	**Conic Packets**	**Conic-1 light**	**Basic**	**Conic Packets**	**Conic-1 light**	**Basic**	**Conic Packets**	**Conic-1 light**	**Basic**	**Conic Packets**	**Conic-1 light**
fps	4.2 fps	16.1 fps	6.2 fps	2.9 fps	7.1 fps	3.3 fps	2.39 fps	3.5 fps	1.8 fps	1.13 fps	1.5 fps	0.62 fps
Ray/Box	29.5 M	2 K	1.9 M	47 M	1.54 M	9.44 M	27.5 M	6.73 M	17 M	66.5 M	5.6 M	17.3 M
Cone/Box	0	1.1 M	2.05 M	0	1.27 M	2.05 M	0	486 K	850 K	0	1.7 M	3.2 M
Triangle/Box	32 K	37 K	68 K	5.63 M	5.7 M	11.1 M	31.7 M	32.3 M	58.9 M	154 M	167 M	419 M
Ray/Triangle	23 M	5.1 M	12 M	18.9 M	9.9 M	20.5 M	10.2 M	10.2 M	19.6 M	15.8 M	14.7 M	23.8 M
Nodes/DAC	888	6 K	8.2 K	42 K	50 K	86.4 K	208 K	212 K	371 K	157 K	171 K	275 K

Fig. 8. Number of intersections and node traversals needed to generate a 1280 × 800 image with the divide-and-conquer algorithm, showing three different rendering scenarios: no optimization, conic packets, and 1 light source.



Toasters:
11K Triangles, 247 frames
14.5 fps, **15.2 fps**, 16.12 fps

Ben:
78K Triangles, 32 frames
11.3 fps, **11.9 fps**, 12.3 fps

Fairy:
174K Triangles, 21 frames
6.1 fps, **6.5 fps**, 6.9 fps

Exploding Dragon:
252K Triangles, 15 frames
4.9 fps, **7.5** fps, 11.7 fps

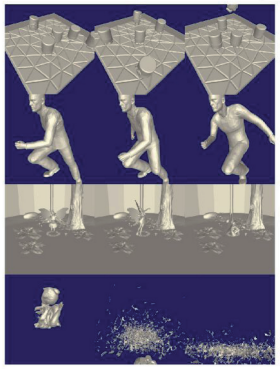


Fig. 9. Frame rate for some standard animated scenes (frustum rays, 1024^2 pixels, basic shading) with min, **average** (in bold) and max frame rates indicated on the bottom left.

point light source is activated, which is logical as the entire algorithm must be run again. It also appears that computing shadows is a slightly less efficient process than computing primary rays, probably because of the less regular distribution of shadow rays in space.

Finally, a hint at performance for some standard animated scenes is given in Figure 9. Three of these scenes (Toasters, Ben, and Fairy) exhibit little change in the frame rate along the animation. Indeed, the geometrical aspect of the scenes remains similar throughout the animation. The Exploding Dragon scene nevertheless shows a different trend as the dragon is exploding, with the frame rate dropping by more than half at some point. While the change in the triangle configuration is extreme (from a coherent object to randomly located triangles), Wald [2007] also reported a drop in

Processor	Threads	Rate	Speed-up	Rate	Speed-up	Rate	Speed-up	Rate	Speed-up	Average Speed-Up
3GHz Core 2	1	2.9M/s	0.77	1M/s	0.91	1.1M/s	0.9	0.82M/s	0.88	0.865
3GHz Core i7	1	3.8M/s	1	1.1M/s	1	1.3M/s	1	0.93M/s	1	1
3GHz Core i7	2	7.2M/s	1.9	2.1M/s	2	2.5M/s	2	1.8M/s	2	1.975
3GHz Core i7	4	12.3M/s	3.2	4M/s	3.6	4.6M/s	3.7	3.2M/s	3.4	3.475
3GHz Core i7	8	14.3M/s	3.8	4.6M/s	4.2	5.5M/s	4.3	3.4M/s	3.7	4

Fig. 10. Performance in millions of rays per second for Path-Tracing (7 diffuse bounces, no Russian roulette, 740× 480 pixels). Core i7 results are obtained with the i7 turbo mode option disabled.

performance of the same magnitude (21 fps for the first frame, 13 fps for the fourth on a 2.6 GHz 8-core system).

4.2 Random Rays

A simple path-tracer [Kajiya 1986] implementing diffuse BRDFs and BSSRDFs [Jensen et al. 2001] has been implemented to analyse the behavior of DACRT with secondary and random rays. Each ray undergoes 7 diffuse bounces (Figure 10, no Russian roulette used). In this section, the models used are ERW6, Sponza, Conference Room, and Soda-Hall.

For path-tracing, results show that the practical performance ranges from 0.9 to 3 million rays per second. While difficult to quantify, this range seems very competitive for purely random rays. For the conference room, DACRT performs at close to 65% of what Tsakok's [2009] highly optimized ray-tracer can achieve after having constructed a spatial subdivision data structure. Obviously, there is a price for not using spatial subdivision data structures, but ray-tracing does not wait for construction and starts immediately, which is useful for animated renderings. Figure 10 also shows results for a parallel version of the algorithm running on a 4 cores/8 threads 3 GHz Intel core i7 with turbo mode disabled. Simple path-tracing parallelism has been achieved by giving a full image iteration to each thread. Each thread also handles separate primitive and ray indexing lists. Results show that the speedup is almost linear and close to 4× with 8 threads running. However, the average speedup for 4 threads is only 3.5×. This could be a sign of limited memory bandwidth, but we are not discounting a problem with the OS scheduler due to hyperthreading [McGuire and Luebke 2009]. We will need to investigate parallelization algorithms more thoroughly in the future, especially on highly parallel platforms like GPUs.

# Bounces	Original Sampling	TT: 158s (7700 VPLs)			TT: 195s (8000 VPLs)			TT: 265s (5200 VPLs)			TT: 127s (5020 VPLs)		
		Photons	t	Rate	Photons	t	Rate	Photons	t	Rate	Photons	t	Rate
1	1K	1.7k	<1ms	NA	1.5K	6ms	250K/s	1.6K	13ms	123K/s	1.6K	22ms	72K/s
	10K	17K	6ms	2.8M/s	15.5K	20ms	775K/s	16.9K	28ms	603K/s	17K	34ms	500K/s
	40K	68K	25ms	2.7M/s	62K	56ms	1.1 M/s	68K	62ms	1.1M/s	68K	79ms	869K/s
3	1K	2.5K	1ms	2.5M/s	2K	15ms	133K/s	2.5K	38ms	65K/s	2.5K	76ms	32K/s
	10K	25K	13ms	2M/s	21.5K	47ms	457K/s	25K	78ms	320K/s	25K	103ms	242K/s
	40K	101K	56ms	1.8M/s	87K	127ms	685K/s	101K	164ms	615K/s	101K	202ms	500K/s
5	1K	2.8K	2ms	264K/s	2.3K	21ms	109k/s	2.9K	59ms	49K/s	2.9K	143ms	20K/s
	10K	29K	15ms	2.2M/s	24.4K	65ms	375K/s	29K	118ms	245K/s	29K	210ms	138K/s
	40K	118K	69ms	580K/s	94K	167ms	562K/s	116K	236ms	491K/s	117K	326ms	358K/s

Fig. 11. Performance for creating a distribution of photons inside a scene for 3 light source samplings and 1, 3, and 5 indirect bounces. Images (740 × 480) are created from such distributions using a basic instant radiosity algorithm, with Tracing Time (TT) and number of photons/virtual point lights (VPLs) indicated above.

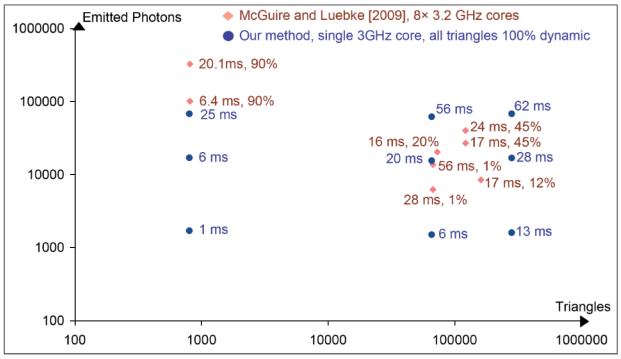


Fig. 12. Comparison of CPU trace time for generating a photon map (1 indirect bounce) according to the total number of triangles in the scene and the number of photons used. Percentages indicate the number of dynamic triangles used in McGuire and Luebke [2009] tests.

While our path-tracing implementation handles quite a large amount of rays (i.e., 740 × 480) per pass and its efficiency can be seen as an upper limit, further assessment of DACRT is needed for more efficient global illumination techniques like instant radiosity [Keller 1997] or image space photon mapping [McGuire and Luebke 2009], which require only a limited amount of photons to be initially distributed in space. The efficiency of this process is demonstrated in Figure 11, where time to generate a photon distribution and time for the overall rendering with instant radiosity are given. Primary ray intersections/photon visibility for this algorithm has been achieved using conic packets. Results show that emitting a small number of photons in real time is possible in many situations. Provided that the rest of the pipeline can be executed in real time, DACRT is an algorithm of choice for photon emissions in dynamic scenes. Of interest is the efficiency in terms of rays per second, which drops very quickly as the number of photons gets low, similar to other dynamic methods if the construction of a spatial subdivision structure is included.

Giving a fair performance assessment/comparison for generating a photon distribution is a difficult task as results depend on both the scene size and the number of photons emitted. For our last test (Figure 12), we made an attempt to do so by merging reference samples taken from McGuire and Luebke [2009], and our results (in Figure 11) for one indirect bounce. It is important to note that

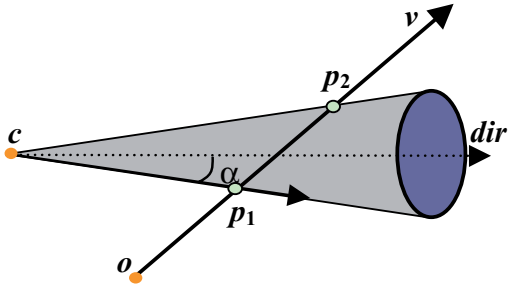


Fig. 13. Line/cone intersection.

the graph excludes most testing conditions except for the scene size and the number of photons. Also, our results are obtained on a single 3 GHz core (8 cores cadenced at 3.2 GHz), with all the triangles considered as dynamic. For small scenes, McGuire and Luebke [2009] approach looks faster as it generates more photons in less time. Nevertheless, timings to generate the respective distributions look closer for larger scenes, and approximately of the same order. As the sampling size is small and testing environments may differ, we only draw the conclusion that our technique looks very promising and has potential for generating photon distributions in dynamic scenes.

5. CONCLUSION

The intent at the beginning of this work was to demonstrate that ray-tracing can be achieved efficiently in terms of complexity without using or storing a spatial subdivision data structure. Divide-And-Conquer Ray-Tracing is a new and simple algorithm that goes even beyond this initial objective as it demonstrates that it is on-par with many techniques in terms of speed on equivalent platforms.

This work has demonstrated the potential of Divide-And-Conquer Ray-Tracing, including the possibility of handling sets of triangles and rays as soups in main memory, quickly solving larger problems, and having a simplified and deterministic memory management. This makes this approach well suited to dynamic scene contexts. This is especially true as the scene becomes larger. Whereas many other approaches for primary ray visibility are no longer interactive when there are more than a few hundred thousand triangles, our approach can interactively process millions of triangles for primary rays on a single core.

All these advantages come, however, at the expense of bundling enough rays per rendering pass to obtain maximum efficiency by maintaining a comparatively low cost for streaming triangles. This is a nonissue as a low number of rays/photons can actually be processed in real time and clever bundling of rays is always possible, but requires specific workload balancing. As no work has previously been carried out on ray-tracing without acceleration data structures, considerable further research is required to optimize this technique, a prospect that makes Divide-and-Conquer Ray-Tracing even more attractive. For instance, though our algorithm can be faster than many multithreaded applications, massively parallel algorithms still need to be researched. Reusing parallel methods similar to the ones used for kd-tree constructions by investigating left and right nodes in parallel may result in nonlinear speedup. This is because some of the rays terminate before the second node is investigated. Thus, some streaming operations are saved when nodes are processed sequentially. Therefore, we have not thoroughly investigated this area as it is a complex problem beyond the scope of this article, and we do not yet have a simple, unique, and optimal solution to

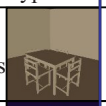
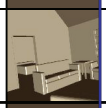
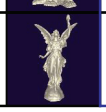
	OpenGL NV 8800 GTX Normal/ Display Lists	Shevtsov et al [2007] Core 2 3 GHz	Lagae & Dutré [2007] X5365 3GHz *w/ faster grid*	**DACRT** Core 2 3 GHz	Wald et al. [2006] BVH Opter. 2.6GHz 16x16 packets	MLRT from [Overbeck et al. 2007] P4HT 3GHz	Overbeck et al. [2007] P4HT 3GHz + ATI rad. 9800	Reference SAH Ray Tracer 8×8 Packet Core 2 3GHz
Cores or Threads used	128	4	1	1	1	1	1 / NA	1
Renderer Type	Dynamic	Dynamic	Dynamic	Dynamic	Mixed	Static	Static	Static
Erw6 804 triangles 	1K/1K fps 65/65 ×			**15.3 fps** 1 ×	71 / 42 fps 4.6/2.8 × *build/render*	74/ 13 fps 4.8/ 0.84 × *quad on/off*	169 fps 11 ×	5.1 fps 0.33 ×
Bunny 69 K triangles 	211/1K fps 22/105 ×		1.44 fps 0.15 ×	**9.5 fps** 1 ×				3.38 fps 0.35 ×
Fairy 174 K triangles 	45/66 fps 6.6/9.7 ×	5.84 fps 0.85 × *tex+2 lights*		**6.8 fps** 1 ×	0.35/6.4 fps 0.05/0.94 × *1st build/next im.*			3.7 fps 0.54 ×
Cabin 217 K triangles 	18/166 fps 1.5/14.4 ×		0.38 fps 0.033 ×	**11.5 fps** 1 ×				4.2 fps 0.37 ×
Conference 282K triangles 	26/333 fps 2.5/32 ×			**10.1 fps** 1 ×	0.2/10.5 fps 0.02/1.03 × *build/render*	10.8/ 5.6 fps 1.25/ 0.65 × *quad on/off*	5.6 fps 0.55 ×	3.6 fps 0.35 ×
Armadillo 345K triangles 	13/333 fps 2.4/63 ×		1.05 fps 0.2 ×	**5.2 fps** 1 ×		7/ 4.3 fps 1.32/ 0.82 × *quad on/off*	1.7 fps 0.32 ×	4.7 fps 0.9 ×
Conference 1.02 M triangles 	7/15.5 fps 0.8/1.8 ×		0.41 fps 0.04 ×	**8.6 fps** 1 ×				2.7 fps 0.31 ×
Buddha 1.08 M triangles 	6.9/15.1 fps 1.6/3.6 ×	2.1 fps 0.5 × *build only*	1.38 fps 0.33 ×	**4.2 fps** 1 ×	0.05/NA fps 0.01 / *NA* × *build/render*			4.2 fps 1 ×
Soda Hall 2.2 M triangles 	3.3/9 fps 0.37/1.02 ×	2.1 fps 0.24 × *build only*		**8.7 fps** 1 ×	0.018/12 fps 0.002/1.4 × *build/render*	16.2/ 7.3 fps 1.84/ 0.84 × *quad on/off*	21 fps 2.4 ×	6.9 fps 0.69 ×
Dragon 7.2 M triangles 	1.03/2.77 fps 0.66/1.77 ×	0.58 fps 0.37 × *build only*	0.4 fps 0.28 ×	**1.56 fps** 1 ×				2.24 fps 1.43 ×
Thai Statue 10 M triangles 	0.67/1.81 fps 0.55/1.56 ×	0.4 fps 0.31 × *build only*	0.36 fps 0.28 ×	**1.28 fps** 1 ×				1.46 fps 1.27 ×
Lucy 28 M triangles 	0.26/*NA* fps 0.57/*NA* ×		0.19 fps 0.43 ×	**0.45 fps** 1 ×				0.73 fps 1.61 ×

Fig. 14. Ray-Casting results (fps and performance relative to DACRT) for 1024 × 1024 images with conic packet optimization. The MLRT [Reshetov et al. 2005] results have been taken from Overbeck et al. [2007] with quad optimization on and off. Our reference kd-tree based packet ray-tracer is fully optimized, while the OpenGL results are given for triangles either sent through the PCI-E bus or stored in display lists.

propose. In future work, we aim to explore GPUs implementations as the low memory requirements of our algorithm seem perfectly suited to GPUs. However, the limitations inherent to such platforms as high latency global synchronization will introduce additional difficulties to overcome.

Finally, this article has been restricted to only one type of spatial subdivision scheme and one type of primitive. Other types of spatial subdivisions like (possibly perspective) grids have already demonstrated specific advantages and disadvantages for the visualization of triangles. Specialized heuristics and the optimal subdivision scheme still remain to be determined, for which we do not yet have an answer. Beyond this article, we hope the general divide-and-conquer scheme can be applied to numerous fields of computer graphics where existing solutions may not be satisfactory.

APPENDIX: CONE/EDGE INTERSECTION

The starting point of our edge/cone intersection solution [Eberly 2000] is that at the intersection location(s), an angle α is made between the main cone direction and a vector defined from the cone center and the intersection point (see Figure 13). The following equation holds.

$$\frac{\overrightarrow{cp_1}}{||cp_1||} \cdot \frac{\overrightarrow{dir}}{||dir||} = \cos(\alpha)$$

Now we can simplify the problem by squaring the two sides of the equation and supposing that the cone direction is normalized. We get

$$((o + v \cdot t - c) \cdot dir)^2 = (\cos(\alpha) \cdot ||o + v \cdot t - c||)^2.$$

Finally, expanding the equations leads to a second-degree polynomial that can easily be solved to get the two intersection parameters.

$$((v \cdot dir)^2 - \cos^2(\alpha) \cdot v^2) \cdot t^2 + (2 \cdot (co \cdot dir) \cdot (v \cdot dir)$$
$$-2 \cdot \cos^2(\alpha) \cdot (co \cdot v)) \cdot t + (co \cdot dir)^2 - \cos^2(\alpha) \cdot co^2 = 0$$

Note that if no real roots exist, then there is no intersection between the cone and the edge. The final step involves checking that the edge range actually falls between the roots, and that the solution is actually on the positive side of the cone. Indeed, squaring first both sides of the equation introduces a double-sided cone problem.

ACKNOWLEDGMENTS

We would like to thank Sally Andrew for proof-reading of the article and the reviewers for their valuable feedback.

REFERENCES

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

AMANATIDES, J. 1984. Ray tracing with cones. In *Proceedings of ACM SIGGRAPH*, 129–135.

BENTLEY, J. L. Multidimensional binary search trees used for associative searching. 1975. *Comm. ACM 18*, 9, 509–517.

CLEARY, J. G. AND WYVILL, G. 1988. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Vis. Comput. 4*, 2, 65–83.

EBERLY, D. 2000. Intersection of a line and a cone. http://www.geometrictools.com/Documentation/IntersectionLineCone.pdf

FUCHS, H., KEDEM, Z. M., AND NAYLOR B. F. 1980. On visible surface generation by a priori tree structures. In *Proceedings of SIGGRAPH*. 124–133.

FUJIMOTO, A., TANAKA, T. AND IWATA, K. 1986. Arts: Accelerated ray-tracing system. *IEEE Comput. Graph. Appl. 6*, 4, 16–26.

GARANZHA, K. AND LOOP, C. 2010. Fast ray sorting and breadth-first packet traversal for GPU ray tracing. In *Proceedings of Eurographics*. 289–298.

GLASSNER A., S. 1984. Space subdivision for fast ray tracing. *IEEE Comput. Graph. Appl. 4*, 10, 15–22.

GRIBBLE, C. P. AND RAMANI, K. 2008. Coherent ray tracing via stream filtering. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 59–66.

HAVRAN., V., HERZOG, R., AND SEIDEL H.-P. 2006. On the fast construction of spatial hierarchies for ray tracing. In *Proceedings of the IEEE Symposium on Raytracing*. 71–80.

HUNT, W., MARK, W. R. AND STOLL, G. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Proceedings of the IEEE Symposium on Raytracing*. 81–88.

HUNT, W., AND MARK, W. R. 2008a. Ray-Specialized acceleration structures for ray tracing. In *Proceedings of the IEEE/EG Symposium on Raytracing*. 3–10.

HUNT, W., AND MARK, W. R. 2008b. Adaptive acceleration structures in perspective space. In *Proceedings of the IEEE/EG Symposium on Raytracing*. 11–17.

IZE, T., SHIRLEY, P. AND PARKER, S. G. 2007. Grid creation strategies for efficient ray tracing. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 27–32.

JENSEN, H. W., MARSCHNER, S. R., LEVOY, M., AND HANRAHAN, P. 2001. A practical model for subsurface light transport. In *Proceedings of ACM SIGGRAPH*. 501–518.

KAJIYA, J. 1986. The rendering equation. In *Proceedings of SIGGRAPH'86*. 143–150.

KAY, T. AND KAJIYA, J. 1986. Ray Tracing complex scenes. In *Proceedings of SIGGRAPH'86*. 269–278.

KELLER, A. 1997. Instant radiosity. In *Proceedings of SIGGRAPH*. 49–56.

LAGAE, A. AND DUTRÉ, P. 2008. Compact, fast and robust grids for ray tracing. In *Proceedings of the Eurographics Symposium on Rendering*. 1235–1244.

LAUTERBACH, C., YOON, S. AND MANOCHA, D. 2006. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 39–45.

MCGUIRE, M. AND LUEBKE, D. 2009. Hardware-Accelerated global illumination by image space photon mapping. In *Proceedings of the Conference on High Performance Graphics*. 77–89.

MÖLLER, T. AND TRUMBORE, B. 1997. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools 2*, 1, 21–28.

NAKAMARU, K. AND OHNO, Y. 1997. Breadth-First ray tracing utilizing uniform spatial subdivision. *IEEE Trans. Vis. Comput. Graph. 3*, 4, 316–328.

OVERBECK, R., RAMAMOORTHI, R. AND MARK, W. R. 2007. A real-time beam tracer with application to exact soft shadows. In *Proceedings of the Eurographics Symposium on Rendering*. 85–98.

ROGER, D., ASSARSON, U., HOLZSCHUCH, N. 2007. Whitted ray-tracing using a ray-space hierarchy on the GPU. Multi-level ray tracing algorithm. In *Proceedings of the Eurographics Symposium on Rendering*.

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

RESHETOV, A. 2006. Omnidirectional ray tracing traversal algorithm for kd-trees. In *Proceedings of the IEEE/EG Symposium on Raytracing*. 57–60.

RUBIN S., WHITTED, J. 1980. A 3-dimensional representation for fast rendering of complex scenes. In *Proceedings of SIGGRAPH*. 110–116.

SCHMITTLER, J., WOOP, S., WAGNER, D., PAUL, W. J., AND SLUSALLEK, P. 2004. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of the Graphics Hardware*. 95–106.

SHEVTSOV, M., SOUPIKOV, A. AND KAPUSTIN, A. 2007. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. In *Proceedings of Eurographics*. 395–404.

TSAKOK, J. A. 2008. Faster fast ray tracing techniques. Master thesis, Electrical and Computer Engineering. University of Waterloo.

TSAKOK, J. A. 2009. Faster incoherent rays: Multi-BVH ray stream tracing. In *Proceedings of the Conference on High Performance Graphics*. 151–158.

WÄCHTER, C. AND KELLER, A. 2006. Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the Eurographics Symposium on Rendering*. 139–149.

WÄCHTER, C. AND KELLER, A. 2007. Terminating spatial hierarchies by a priori bounding memory. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 41–46.

WÄCHTER, C. AND KELLER, A. 2009. Efficient ray tracing without acceleration data structure. U.S. Patent Applications Publication No. US 2009/0225081 A1.

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

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006a. Ray tracing animated scenes using coherent grid traversal. In *Proceedings of ACM SIGGRAPH*. 485–493.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2006b. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1, Article 6.

WALD, I. 2007. On fast construction of SAH based bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 485–493.

WHITTED, T. 1980. An improved illumination model for shaded display. *Comm. ACM 23*, 6, 343–349.

ZHOU, K., HOU, Q., WANG, R., GUO., B. 2008. Real-Time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5, Article 126.

Received May 2010; revised February 2011; accepted xxxx