Multi-Level Ray Tracing Algorithm

Alexander Reshetov Alexei Soupikov Jim Hurley
Intel Corporation

Abstract

We propose new approaches to ray tracing that greatly reduce the required number of operations while strictly preserving the geometrical correctness of the solution. A hierarchical "beam" structure serves as a proxy for a collection of rays. It is tested against a kd-tree representing the overall scene in order to discard from consideration the sub-set of the kd-tree (and hence the scene) that is guaranteed not to intersect with any possible ray inside the beam. This allows for all the rays inside the beam to start traversing the tree from some node deep inside thus eliminating unnecessary operations. The original beam can be further sub-divided, and we can either continue looking for new optimal entry points for the sub-beams, or we can decompose the beam into individual rays. This is a hierarchical process that can be adapted to the geometrical complexity of a particular view direction allowing for efficient geometric anti-aliasing. By amortizing the cost of partially traversing the tree for all the rays in a beam, up to an order of magnitude performance improvement can be achieved enabling interactivity for complex scenes on ordinary desktop machines.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism — Ray Tracing, Global Illumination, Beam Tracing, Geometric Anti-Aliasing.

Keywords: ray-tracing, frustum occlusion culling, anti-aliasing

1 Introduction

A renewed interest in Ray Tracing (RT) algorithms is being fueled by the relentless progress of Moore's law in terms of raw compute power and various algorithmic discoveries resulting in significant performance improvements. This makes real-time Ray Tracing and Global Illumination (GI) attractive for implementation on desktop machines. Some of these new discoveries are summarized by Wald et al. [2003]. Table 1 provides comparison results for our implementation and those described by Wald. We were able to improve performance further by up to an order of magnitude. This is achieved by amortizing per-beam operations which would otherwise be performed for each ray in a group.

e-mail: {Alexander.Reshetov, Alexei.Soupikov, Jim.Hurley}@intel.com

Copyright © 2005 by the Association for Computing Machinery, Inc.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212) 869-0481 or e-mail permissions@acm.org.
© 2005 ACM 0730-0301/05/0700-1176 $5.00

In this paper, we focus on the most fundamental task in ray tracing, namely, finding the intersection of one or more rays with a given geometry. We consider reducing the average number of operations per ray as the most objective metric for comparing different algorithms. Generally speaking, there is no one guaranteed best ray tracing algorithm as performance depends on many factors such as: the overall scene complexity, the current view of the scene, and the characteristics of the host platform. Many of the techniques we use have been known for some time, although not specifically applied to the problem of efficient ray tracing. For example, a modified view frustum culling algorithm is used to reduce redundant operations for big groups of rays. In essence, we're building on top of the work of Heckbert and Hanrahan [1984], Teller and Alex [1998], Assarsson and Möller [2000], Kay and Kajiya [1986], and others. Another motivating factor is to address the issue of geometrical aliasing which is especially pronounced at interactive rates. One way to improve the quality of the resulting image is to cast multiple rays through each pixel. We propose a mechanism for dynamically measuring the geometrical complexity for a given view direction, which can be used for budgeting rays more cost-effectively.

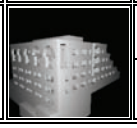
Framerate (FPS) @ 1024x1024 resolution / scene # of triangles and shader (+/-)			OpenRT @ 2.5 GHz P4 1 thread	MLRTA @ 2.4 GHz P4 1 thread	MLRTA @ 3.2 GHz P4 with HT 2 threads
Erw6 804		– shader	7.1	70.2	109.8
		+ shader	2.3	37.8	50.7
Confe-rence 274K		– shader	4.55	11.2	19.5
		+ shader	1.93	9.5	15.6
Soda Hall 2195K		– shader	4.12	21.1	35.5
		+ shader	1.8	15.3	24.1

Table 1: Framerate comparison results for 3 scenes. OpenRT data is taken from Wald et al. [2003]. Two sets of data are provided: one for a null shader, and the other for a simple shader (equivalent to placing a point light at the camera position).

We start by giving a brief overview of related work in section 2. Section 3 introduces the basic concepts which will be used throughout this article. By following relevant prior work, these concepts will be described and then refined and adapted for our approach. Then we describe the complete multi-level ray tracing pipeline in section 4 and discuss results in section 5. We conclude with some considerations for further research ideas in section 6, which will also describe limitations of the proposed method.

1.1 MLRTA Overview

The central contribution of this paper is a new robust approach to high performance ray-tracing which is achieved without any

approximations or geometric simplifications. Instead of looking for ray/geometry intersections, we effectively *exclude* certain objects from consideration for any given group of rays. This is accomplished by providing a separate entry point into the global acceleration structure (kd-tree) for each group of rays. Instead of traversing each constituent ray from the top of the tree, we start at the group's entry node inside the tree. This reduces the average number of traversal steps by 2/3^{rds} for typical scenes.

Further insight came from analyzing the frustum culling algorithms used for axis-aligned bounded boxes (AABB) in the context of traditional raster graphics. Here, simple tests are used on AABB proxies for detailed geometries to purge objects which do not intersect the frustum, effectively reducing the amount of geometry submitted for rendering. These tests are designed to quickly detect the majority of cases where objects do not intersect with the view frustum. The objects can be trivially rejected if their AABBs can be separated from the frustum by one of the frustum's planes. This works well in those cases where small AABBs are culled by a relatively big frustum. A comparable approach may be used in ray tracing, where the AABBs represent the volumes associated within a kd-tree's nodes, and the frustum represents a beam of rays. Typically, this frustum will be much smaller than most of the AABBs in the kd-tree. As a result, using the trivial reject algorithm as described above would result in a higher percentage of redundant potential accepts (cases where we cannot trivially exclude the intersection for non-intersecting objects). Conversely, for RT applications we propose reversing the roles of the frustum and AABB by using the AABB faces as separation planes during depth-first traversal of the kd-tree.

Furthermore, we recommend new approaches to the kd-tree building process which make it more suitable for the Multi Level Ray Tracing Algorithm (MLRTA) proposed herein. We also introduce the new concept of an "empty occluder", which is basically a tagged empty box contained inside a watertight object. We use such empty occluders to stop further traversal of beams that completely intersect such occluders.

All these innovations together allow us to improve the performance of the ray-tracing algorithm by up to an order of magnitude compared with previous results in the literature. As an added bonus, the MLRTA provides a natural mechanism for measuring the geometric complexity of the portion of the scene visible to a given group of rays, which enables geometric anti-aliasing.

2 Related Work

A number of researchers have developed strategies for exploiting coherence between spatially adjacent rays. We took inspiration from the early work of Heckbert and Hanrahan [1984], in which polygonal beams were used instead of rays to improve the anti-aliasing properties of an image. The beam was annotated as it intersected with objects in the scene so that the edges of these interfering objects could be more effectively anti-aliased. Similar goals were pursued by different researchers through the introduction of cone tracing [Amanatides 1984], pencil tracing [Shinya at el. 1984], and ray bounds [Ohta and Maekawa 1990].

In [Heckbert and Hanrahan 1984], the beams are persistent with extra information accumulated during tracing to describe multiple beam/object intersections. Later, researchers switched to splitting beams whenever such intersections occurred. In [Ghazanfarpour and Hasenfratz 1998], this happens when a beam intersects multiple objects thus necessitating smaller sub-beams to precisely anti-alias a polyhedral scene. In [Genetti et al. 1998], "pyramidal rays" (pyrays) are split when any part of one intersects an object. Arvo and Kirk [1987] use a volume in a 5D space to represent a collection of rays (3D for origin and 2D for direction). The original 5D volume is then dynamically subdivided into hypercubes, each linked to a set of objects which are candidates for intersection. In [Heckbert and Hanrahan 1984], the beam tree which represents the surfaces intersected by the beam, is computed in object space and then passed to a polygon renderer for scan conversion. In [Teller and Alex 1998], frustum casting is proposed which samples discretely in image space, but operates in object space. In this algorithm, the frustum is recursively subdivided, while object space is organized linearly with indices identifying neighbors of a given current cell. In [Cho and Forsyth 1999], a visibility complex is incrementally constructed enabling efficient ray/geometry queries.

In [Havran and Bittner 2000], Longest Common Traversal Sequences are used to amortize common operations among multiple rays. In [Dmitriev et al. 2004], pyramidal shafts are used for the same purpose. This technique is also extended to secondary and shadow rays. In both of these works the convex hull of a group of rays is represented by a few boundary rays, which are traversed through the scene. It is then assumed that all interior rays will follow the same path. This is not always correct and we will provide examples illustrating this point in section 3.3. We believe that we have come up with the necessary mathematical apparatus which is geometrically accurate and achieves the same goals.

All these algorithms attempt to combine view-dependent culling in object space with some distance-based visibility determination in image space. This is generally achieved by the use of a spatially distributed recursive construct which initially encompasses multiple rays and is then progressively refined. We use the same approach, the fundamental difference between our method and these others is that we are not trying to find objects that this construct intersects. Instead, we eliminate those objects that *do not* intersect with the construct.

A different category of algorithms aims at minimizing the required number of intersection tests by budgeting rays diligently, sampling sparsely in areas of low geometric variation and super-sampling for geometrically complex or perceptually important parts of the image [Ramasubramanian et al. 1999]. Our approach facilitates this type of optimization by providing a natural measure of geometrical complexity for a specific viewpoint.

3 Basic Concepts

3.1 Acceleration Structures

To naively find an intersection of a ray with a scene, one could test this ray against all objects in the scene for an intersection and keep the one with the shortest distance from the ray origin. This algorithm might have the lowest memory footprint, but its execution time is prohibitive. A much better approach would be to organize the scene into some sort of data structure (usually called an "acceleration structure" – AS) and use this structure to zero in on the area of interest in a hierarchical fashion. An AS works by splitting 3D space into subsets containing a certain number of primitive objects (triangles if no other primitives are used). In addition to this spatial organization, specific traversal routines are defined as well. These routines are used to quickly decide which

subsets to look into further for possible intersections. Different traversal routines may coexist for the same AS, for example, one for primary and another for shadow rays. One object may belong to multiple subsets and some subsets may be empty. Various types of ASs that lend themselves to different scene geometries and/or computer architectures are well known and described in the literature [Szirmay-Kalos et al. 2002]. In this work we use a kd-tree data structure; a systematic analysis of the kd-tree building algorithm was first given by Glassner [1984] and followed by numerous publications, in particular by Havran [2000].

The main operation in the kd-tree building process is to split an axis-aligned bounding box into two (potentially unequal) boxes by a plane orthogonal to one of the axes. The process is repeated recursively until some termination criteria are met. The splitting algorithm and termination criteria, in essence, define a particular flavor of a kd-tree building algorithm. The split position is chosen by minimizing a cost function over a set of candidate split positions, such as the coordinates of vertices inside the cell and the coordinates of triangle/cell intersections. Following MacDonald and Booth [1990], we use a surface area-based cost function which is computed by multiplying the area of the cell by the number of objects intersecting with it. We have modified the pure area-based approach to bias it in favor of creating large empty cells and also 2D cells (cells with a zero extent along one of the axes). The rationale for this adjustment is that a pure area-based cost function underestimates the importance of placing such cells closer to the top of the tree. This modification is based on three simple rules applied sequentially:

1. **We always create an empty cell if its volume with respect to the original cell is greater than some threshold (10% in our implementation).**
2. **If there is a possible split plane which is completely covered by co-planar triangles, it will be selected and these triangles will be included in the smaller sub-cell. This heuristic sits well with the axis-aligned nature of kd-trees.**
3. **In addition to termination criteria based solely on a cost function (cost of splitting > cost of non-splitting), we also avoid creating very small cells, as measured by the ratio of the cell area to the area of the bounding box of the scene.**

The motivation for rule (3) is to avoid cells that may be so small that it is unlikely to be hit even by one ray for a given camera position from which the whole scene is visible. These rules also speed up the kd-tree building process because, once conditions 1-3 are established, no further processing is necessary.

These rules have to be considered together with a traversal routine. For example, if a traversal routine cannot handle 2D cells, the second rule is not feasible. In our opinion, there is no guaranteed best kd-tree building or traversal algorithm suitable for all scenes or all computer architectures. For current PCs and our implementation of the traversal algorithm (see section 4.3), these rules yielded roughly a 50% improvement in traversal performance compared with pure area-based approaches.

In section 4.1 we will continue the discussion of kd-tree creation and traversal algorithms by assessing the validity of the basic concepts used and analyzing their drawbacks. This leads us naturally to the concept of multi-level ray tracing. But before doing that we have to discuss some additional concepts.

3.2 Grouping Rays Together

The evolution of desktop PCs (both CPUs and GPUs) is such that math operations are getting faster at a higher rate than memory operations. The Single Instruction Multiple Data (SIMD) capability of such devices makes it possible to perform calculations on four rays for the cost of one [Wald 2001; Benthin et al 2003]. This is possible if we carefully choose which rays to shoot together, as there is tremendous geometric spatial coherence to exploit (especially in primary rays). This makes the caching mechanism of modern computers very effective. The performance gains correlate with the size of the group, however, current SIMD hardware can only support four simultaneous operations. What we would like is an algorithm that works independently of hardware features and scales gracefully to a much larger number of concurrent rays.

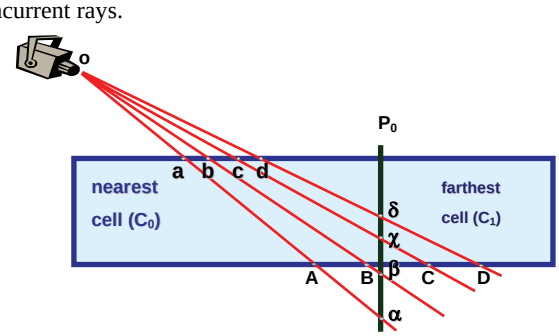


Figure 1: Tracing rays together: different rays go through different cells in the tree.

With grouping of rays comes the necessity to deal with the situation when rays go through different paths in the tree. Indeed, as evident from **Figure 1**, rays **oa** and **ob** travel only through the nearest cell (C_0), while other rays (**oc** and **od**) go through both cells. Obviously, as the number of rays in a group increases, so do the chances that these rays will diverge at some stage in the traversal process. Basically, we have 2 possibilities: we could work with a variable number of rays and regroup them every time some rays "get lost"; or we can mask the inactive rays in the group. Clearly, under such circumstances we can't realize the maximum benefit of processing multiple rays simultaneously, and worse, the overhead of tracking valid rays might even result in inferior performance of the concurrent implementation relative to one that process each ray individually. This also highlights another reason behind some aspects of the termination criteria (3) from the previous section. It doesn't make sense to continue splitting a cell for which only a few rays will still be active in most situations. Effective traversal of groups of rays is possible when all rays follow substantially the same path through the tree. Consequently, the rays must intersect any given split plane in one direction (either going from the negative to the positive direction or vice versa). This translates into the requirement that the coordinates of the direction vectors for all rays in a group have the same sign [Wald et al. 2003]. Groups of rays which do not possess this property must be split into multiple sub-groups which do, or into individual rays.

3.3 Frustum Culling

From the previous section, we see that there can be a penalty for grouping rays together (in the form of redundant operations), and this penalty increases with the size of the group. The desire to avoid or minimize this penalty led us to the development of the multi-level ray traversal algorithm. Even though it looks

tempting, we cannot always use just a few specific rays to ascertain the behavior of the whole group. For example, we may consider using only the 4 corner rays in **Figure 3a** to represent all the rays by proxy. This is the approach utilized by [Havran and Bittner 2000] and [Dmitriev et al. 2004], where boundary rays are used to determine the behavior of internal rays. Unfortunately, this can lead to incorrect traversal choices. Consider the example in **Figure 2**. The top AABB is split into the red and blue sub-cells. We choose two points, **b** and **c** on two faces of the blue cell and one on the edge of the red cell (**a**) as shown below. If we place a camera **o** inside the plane passing through these 3 points, then all 4 of the corner rays **ob**, **oc**, **oe**, and **od** intersect only the blue cell. However, the ray **oa** (located inside the convex hull of the 4 corner rays) passes through both sub-cells. By scrutinizing this example, it is also obvious that even rays strictly inside the convex hull may not always follow the same path as boundary rays.

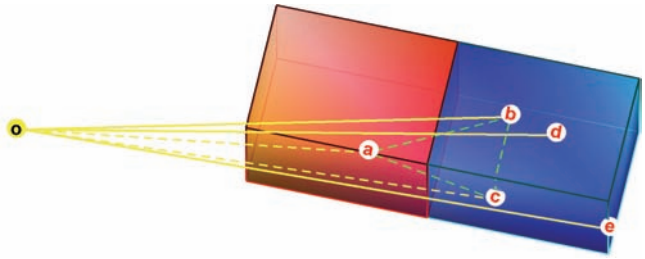


Figure 2: Problems using boundary rays as a proxy for the whole group. The ray **oa**, which is inside the convex hull of the 4 corner rays **obcde**, intersects both cells, while all 4 corner rays intersect only the blue cell.

What we can do instead, is to derive some inclusive properties to characterize the group as a whole, and then use these properties to determine the behavior of the whole group. One intuitive way would be to use the convex hull of all rays in the group, formed by a few given planes. The key operation will then be the determination of whether the convex hull intersects a particular axis-aligned box. This is analagous to the classic frustum culling algorithm used in raster graphics.

We will first describe the straightforward implementation of this algorithm as defined by Assarsson and Möller [2000] and then consider its applicability for ray-tracing purposes. In **Figure 3a**, a frustum is formed by 4 planes intersecting at one point. Each plane is defined by this intersection point and a normal, which we consider to be pointing outward from the frustum. We have to decide whether this frustum intersects with a given axis-aligned box. For each frustum plane, there are two vertices of interest belonging to the axis-aligned box: the one laying farthest in the positive direction of the plane's normal (p-vertex); and the other laying farthest in the negative direction of the normal (n-vertex). By inserting the n-vertex into the plane equation, we can decide whether the n-vertex and therefore the whole box is located outside this particular plane, and so on for each plane, hoping for a trivial rejection. In addition, we could use the p-vertices to determine whether or not the entire box is completely inside the frustum. This approach works for any number of planes forming the frustum.

In the special case where there are exactly 4 planes, the performance of this algorithm can be greatly enhanced by storing the positive and negative components of the 4 normal vectors separately in SIMD form for each of the x, y, and z components. As an axis aligned box can be represented by a pair of extremal vertices (one with the minimum x, y, z, and the other with the maximum x, y, z coordinate values), we can avoid having to find the n-vertices explicitly by noticing that for any frustum plane, the n-vertex has the lower coordinate value (among the 2 extremal vertex possibilities) for any positive normal coordinate, and vice versa for the negative normal direction. Consequently, using hardware with a 4-wide SIMD engine, the box can be culled against all 4 frustum planes using only 6 multiplications and 5 additions, this makes this approach very attractive for performance reasons. If we use "+" and "*" to represent 4-way SIMD operations, these calculations can be performed as follows:

```
__m128 nplane; // 4 plane values
nplane = (plane_normals[0][0] * bmin[0]) +
         (plane_normals[1][0] * bmax[0]);
nplane = (plane_normals[0][1] * bmin[1]) +
         (plane_normals[1][1] * bmax[1]) + nplane;
nplane = (plane_normals[0][2] * bmin[2]) +
         (plane_normals[1][2] * bmax[2]) + nplane;
```

plane_normals[0] and **[1]** contain positive and negative components of the 4 normal vectors and **bmin/bmax** – are the replicated x, y, and z coordinates of the min/max vertices of the current cell. Effectively, the **plane_normals** variable is used as a selector for the appropriate **bmin/bmax** values. With these 11 operations, we manage to find 4 n-vertices and compute all 4 plane values (for each frustum plane). The variable **nplane** now contains these 4 plane values. If any one of these is positive, then the frustum and the box are separated by the appropriate plane.

This trivial rejection mechanism is not perfect, as shown in **Figure 3b**. Here we see the axis-aligned box on the lower left of the frustum as viewed along its center axis. It fails the n-vertex outside test for each of the frustum's planes, implying that it may intersect the frustum, when, in fact, it is entirely outside the frustum. The proportion of these failed trivial rejects increases as the AABB becomes larger and larger (with respect to the frustum's cross section). This problem becomes acute when applied to ray-tracing traversal scenarios where the frustum is used as a proxy for a group of rays. It is often much smaller than the individual AABB cells that it is being tested against. One way to handle this is to reverse the roles of the frustum and AABB: we could use the AABB's planes to attempt to separate it from the frustum instead of the other way around. There is still the danger of failed trivial rejects, but their proportion will be lower due to the favorable ratio of the AABB's cross section to the frustum in the region where they pass one another. One circumstance that makes this tactic especially appealing and easy to implement is when the frustum contains only "coherent" groups of rays, that is, those in which all ray directions have the same sign.

This inverse approach is illustrated in **Figure 3**, (c) and (d). We are trying to decide whether the frustum intersects the red only, the blue only, or both sub-cells formed by the split plane **abcd**, with equation (**x = 1**). Suppose further that the frustum/plane intersection is bounded by the values [**p, n**] for the **y** coordinate, namely that all y-coordinates of the intersection lie in this range. If **n** is less than the y-value for edge **ad**, then we can conclude that:

1. **If all rays have negative Y direction components, the frustum does not intersect the blue sub-cell (Figure 3c). Since we know the frustum intersects the parent cell, we deduce that only the red sub-cell must be traversed.** (1)

2. **If all rays have positive Y direction components, the**

frustum does not intersect the red sub-cell (Figure 3d). Since we know the frustum intersects the parent cell, we know that only the blue sub-cell must be traversed.

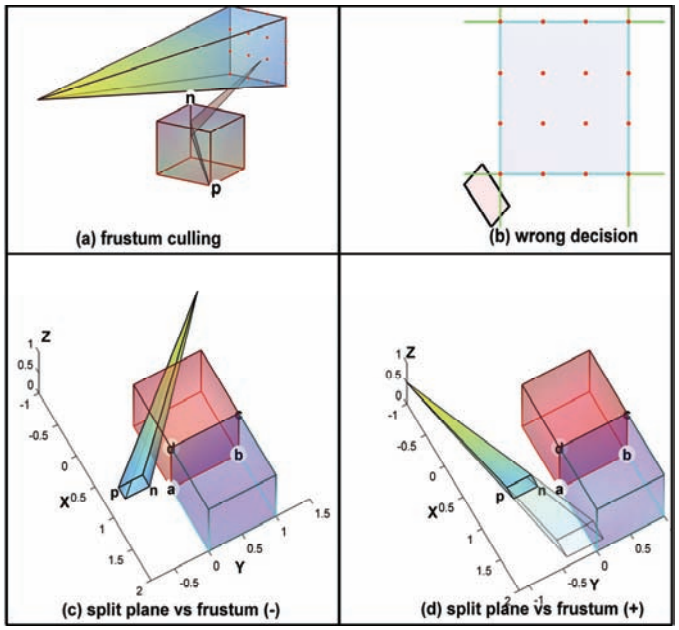


Figure 3: Direct (a,b) and inverse (c,d) frustum culling algorithms.

On architectures which allow it, we can execute these comparison operations in one command, i.e. by comparing the four **y** coordinates of frustum/plane intersection with the y-coordinate of the **ad** edge.

The inverse algorithm does not use frustum plane normals per se. What we need instead is the rectangular bounds for each axis-aligned plane which enclose frustum/plane intersection. This makes it possible to generalize this algorithm for groups of rays which do not have a common origin.

Another important property of the inverse frustum culling algorithm is that when used together with kd-tree traversal, it uses values only for one axis at a time. As we descend into the kd-tree, different axes will be processed at each level, thus allowing for effective culling of the current cell. As with the forward frustum culling algorithm, there may be situations where we erroneously conclude that the frustum might intersect a cell when in fact it does not. Any subsequent processing steps after this point will be wasted. We are primarily concerned with unnecessary intersection tests as they are quite expensive. Most of these extra tests can be avoided by using frustum/plane intersection data as specified in the inverse algorithm and executing additional clipping tests at the leaf nodes. At every leaf node, we perform robust clipping calculations for all 3 possible pairs of axes (**xy**, **yz** and **xz**) against the 6 AABB box faces.

We will illustrate this in **Figure 4** for the case of the **x** and **y** axes by projecting everything on the (**z = 0**) plane. There are 8 possible cases which can be easily detected. These cases differ by the direction of the frustum along the x and y axes and whether the frustum lies above or below the axis aligned box (using z projections). These tests enable us to exclude the great majority of the non-intersecting cases, and even though the remaining ones may cause some redundant calculations, they are tolerable. Amazingly enough, all 8 cases can be tested with only 2 comparisons. We will use terminology and ideas from Kay and Kajiya [1986].

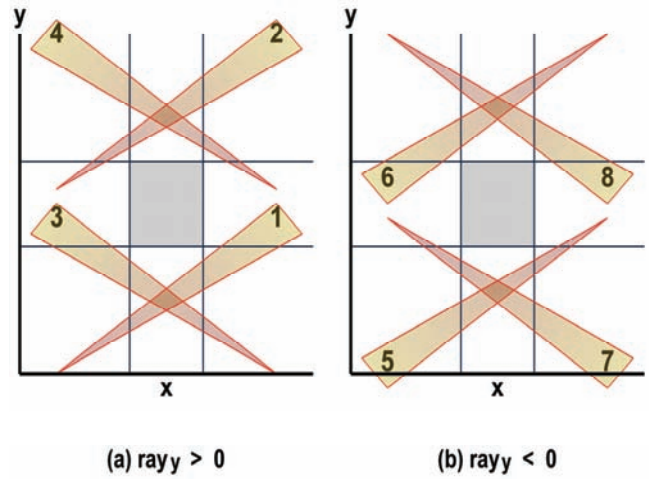


Figure 4: Frustum culling against leaf cell.

An axis-aligned box is defined as an intersection of 3 slabs, where a slab is the space between two parallel planes. For each ray, we may compute the entry and exit points for all 3 slabs which are represented as distances along the ray from the ray's origin. Accordingly, for the whole frustum we will need to know the ranges of these values. If either of the following two statements is true we conclude that the frustum and the box are separated:

1. **(minimum of y-entry values) > (maximum of x-exit values)**
2. **(minimum of x-entry values) > (maximum of y-exit values)** (2)

The first condition here describes cases 1, 3, 6, and 8; the second condition describes the rest. Instead of these two comparisons we could use the direct frustum culling algorithm or even an exact frustum/box intersection test, but this is expensive relative to the number of non-intersection cases that it eliminates. It is also possible to execute these 2 tests (for each pair of axes) at each traversal step, but it is not very effective. We have found that the best approach is to use assessments (1) for the internal nodes followed by (2) for the leaves.

4 Tracing Rays at Multiple Levels

Now we have the mathematical apparatus ready to describe the multi-level ray tracing algorithm (MLRTA). Any hierarchical acceleration structure imposes a certain spatial organization which is then stored in essentially linear memory. During a spatial query this structure is used to find ray/geometry intersection points. Processing is executed by sequentially narrowing the area of interest until final tests resolve the query. As an example from other field of study, we may look for a particular book in a catalog by narrowing the focus from 'science' to 'computer graphics' to 'ray tracing'.

One problem with using kd-trees for 3D scenes is that one doesn't necessarily end up with what might be expected. Consider **Figure 5a**, the box in the middle of the text objects ends up being sub-divided as shown in **Figure 5b**, and it ends up buried deep in the tree because of higher level split decisions made based on the surrounding geometry. If this box became the subject of interest for the camera we would have to traverse the entire kd-tree from

its root for every ray we wish to trace until the box's cells are found (as depicted in **Figure 5c**). Wouldn't it be much better if we could dynamically find alternate entry points in the tree (depending on the current camera view) and start traversing at these nodes? If we are lucky, we might even find that the optimal entry point is right at the leaf node itself. Even better, if we use the beam concept described earlier, we can potentially "traverse" the kd-tree for all the rays in the beam, directly to the leaf node (in this optimistic example) in a single step.

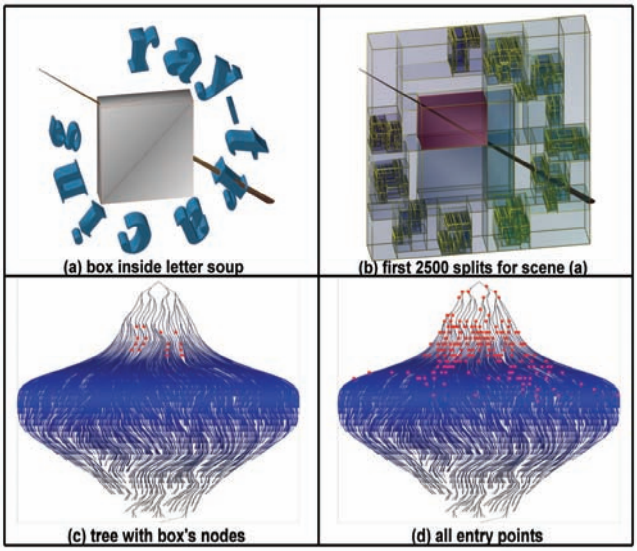


Figure 5: Simple object inside complex scene.

4.1 Finding Ideal Entry Points for Groups of Rays

Restating the results from section 3.3, the following information is sufficient to execute the inverse frustum culling algorithm:

1. **For any given axis-aligned plane, we compute a rectangle inside this plane, which contains all possible ray/plane intersection points. This rectangle does not have to be tight.** (3)
2. **All rays go in the same direction (i.e. x, y, and z projections of ray directions have the same sign).**

In this section we will describe a new algorithm which uses this information and the frustum culling algorithm to find an optimal entry point for all rays that satisfy condition (3). We will refer to this as an "entry point search", or EP search. It will be compared with the traditional depth-first search for intersection points for groups of rays using kd-trees [Wald 2004]. We will refer to this as an "intersection point search" or XP search. The main differences are that the new EP algorithm is not exhaustive and quickly aborts branches of the kd-tree which would not contribute to the final result. The EP search is used to find an optimal place to begin the traditional XP search.

Definition (3) does not presume any particular organization of the rays inside the group. Moreover, it does not even require that we know the specific rays, or even the number of rays, in advance, a fact which will be very handy later when we discuss adaptive tile splitting. The next algorithm describes the steps executed while looking for a common entry point. This is essentially a depth-first traversal of the visible nodes in the tree allowing for early escape from the traversal of branches that will not contribute further to the final result. It is achieved by maintaining a stack of nodes which can be potentially used as entry points (which we will call the "bifurcation stack").

1. **The tree is traversed in depth-first order by**
   - **Using the frustum culling algorithm.**
   - **Store all bifurcation nodes (those where both sub-cells are traversed) in a (last-in-first-out) stack structure until the first leaf node with potential intersections is found. This node is then marked as a candidate entry point node.**
2. **Continue depth-first walkthrough starting from the top-most node on the bifurcation stack. If another leaf with potential intersections is found, the node taken from the bifurcation stack will become the new candidate.**
3. **The algorithm ends and the current candidate node is returned as an entry point if** (4)
   - **The bifurcation stack is empty.**
   - **All potential rays end inside the current leaf node. Two cases are possible:**
     - **The leaf has some objects, and, all rays satisfying condition (3) intersect one of these objects inside the cell.**
     - **The leaf is empty, but is located *inside* some "watertight" object and all rays or groups satisfying condition (3) intersect the bounding box of this leaf.**

While executing this algorithm we are not interested in finding specific ray/object intersections and we may not even be able to do so since the rays in the group are not required to be defined at this point. What we are looking for is the *potential* for intersections. Specifically, if we cannot *exclude* an intersection with any ray satisfying condition (3) we will consider it as a potential intersection. We will illustrate this algorithm using the tree pictured in **Figure 6**.

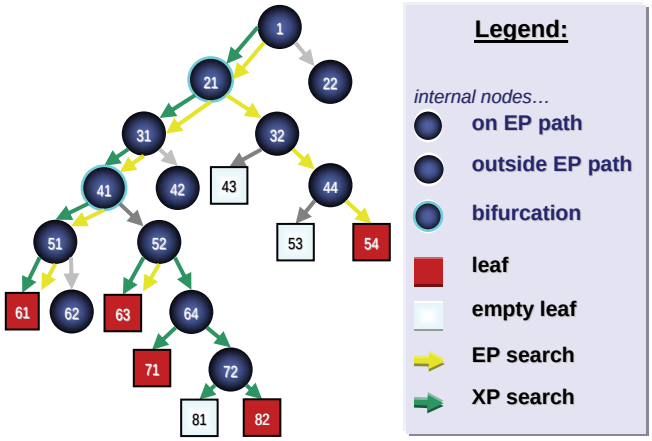


Figure 6: Two traversal algorithms: searching for an entry point (**EP**) and looking for intersections (**XP**).

Starting at node 1 and using group values described by condition (3), we realize that only the left sub-cell 21 has to be traversed. Both sub-cells of 21 have to be considered, so node 21 is stored in the bifurcation stack and processing continues with nodes 31 (split is ignored), 41 (stored in the stack), and 51 (split is ignored). While processing leaf 61, we conclude that there is a potential for intersections. Leaf 61 is then marked as a candidate and the bifurcation stack is frozen. The next node to consider, 41, is taken

from the stack and we continue the depth-first traversal with nodes 52 and 63. Leaf 63 has a potential for intersections, so we mark node 41 as a candidate and move to the next node from the stack (21), abandoning the processing of the sub-tree starting at node 64. From node 21 we go to node 32 which has two children: 43 (ignored because it is empty); and 44 (taken). Node 44 has two leaves: 53 (ignored); and 54 (judged to have potential intersections). Therefore, we will mark node 21 as a candidate and return it as the entry point since the bifurcation stack is empty. All the rays which are bound by condition (3) may now start the tree traversal at node 21.

After the optimal group entry point is found we may split the group and continue looking for better entry points for each sub-group or perform intersection tests for all of the sub-groups to completion (XP search). Note the dissimilarities between the two traversal algorithms, one being a search for a common entry point (EP) and the other which is a search for intersection points (XP):

- At node 21, the XP algorithm is able to exclude node 32 from further processing since all the rays (in the beam) are now known and they intersect only with the cell of node 31. The EP algorithm uses only frustum properties that might intersect with node 32 and therefore traversal of both nodes on behalf of the group is required.
- The EP algorithm ignores node 64 when it reaches its parent, node 52, because it would have no effect on the selection of the group entry point. The XP algorithm however must continue the traversal of nodes 64, 71, 72, 81 & 82 because there may be some intersections to be found there.

This ability of the EP algorithm to disregard non-contributing branches and to do this on behalf of all the rays in a group helps it to greatly reduce the overall computations otherwise performed per ray.

4.2 Tile Splitting

The EP algorithm finds an optimal common entry point for all rays in a group by representing the group as a whole with the ranges of the individual ray directions. For beams representing broad ranges, the EP algorithm will most likely quickly detect that a good (i.e. deep) entry point is not available. In this case, it would be advantageous to split the ray direction ranges (and, accordingly, the actual underlying rays) in an attempt to get better separate EPs for individual sub-groups. Although the rays themselves do not have to be evenly distributed, in our implementation they are.

For primary rays it is practical to choose the initial groups of rays by splitting the image plane into equal tiles. This allows for trivial computation of bounding rectangles for the group/plane intersections as required by (3). Moreover, each tile may be assigned to a separate thread or task on a multi-processor or multi-threaded machine.

There are many possible approaches that could be taken here. In our experiments we used the simplest possible logic to decide what to do with a tile, basing the decision on 3 parameters:

1. **Initial Tile Size (ITS).**
2. **A Minimum Tile Size (MTS) which automatically triggers XP search.** (5)
3. **A Split Factor (SF), which defines how many pieces to split a tile into.**

Any tile larger than MTS is always split unless the chosen entry point is already a leaf. This is illustrated in the sequence of images in **Figure 7**. It is easy to see how the tile sizes change to adapt to the changing underlying geometry. Tiles marked in red are those which have diverging rays (going in different directions) and are therefore unsuitable for the EP search. For such the XP search is performed right away for all the coherent sub-groups of the original tile.

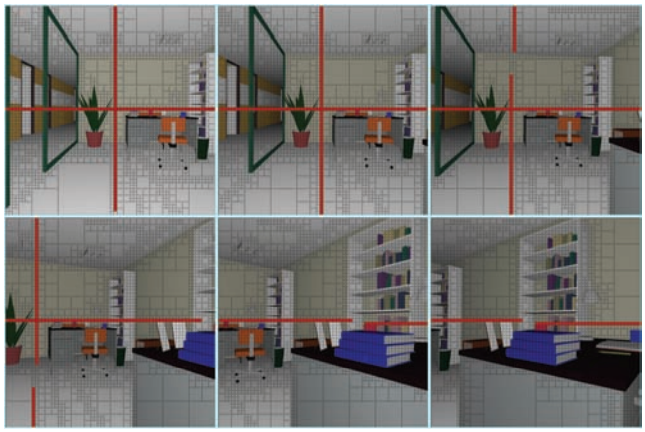


Figure 7: Sequence of frames from Soda Hall scene showing adaptive tile splitting (red color marks diverging packets).

Depending on a scene and/or the camera position, varying the values for the ITS/MTS/SF parameters produce different results. We tested various combinations of these parameters on a collection of over 2500 different models. We varied the Initial Tile Size parameter from 16x16 pixels to 256x256 pixels. Each initial tile was subsequently split either into 4 or 16 sub-tiles and we used a Minimum Tile Size between 4x4 and 64x64 pixels. The measured performance variations were not very large, mostly around 10%. This indicates that it is possible to derive a single set of parameters which would be roughly optimal for most scenes. Our best results were achieved by starting from tiles with 128x128 pixels and splitting them as needed into 16 sub-tiles directly. It is expected that for multi-threaded implementations, a smaller initial tile size might be better because the thread execution granularity will be a function of the time required to process one tile.

Another potential advantage of using the EP algorithm for such tiles is that it provides a natural way to measure geometric complexity. Given the way that the kd-tree creation algorithm works, the size of the sub-tree underneath any given node (measured by overall number of nodes in the sub-tree) serves as an indication of the geometrical complexity in this volume of space. In the extreme case, when an EP is a 2D leaf completely covered by triangles, we could cast rays sparingly and reserve more of the ray budget for more complicated regions. This approach is similar to one described by Ghazanfarpour and Hasenfratz [1998].

4.3 Interval Traversal Algorithm

The basic premise of MLRTA is that it works even if the exact rays in the group are not known. If, however, such information is available, MLRTA can be executed in parallel with the computation of ray/geometry intersection points. We can then use the found distances to intersection points to further purge traversal steps based on visibility culling. In this section we will describe one such implementation which we are currently using in lieu of low level traversal (XP search) in our system. It will be described

together with a sample implementation which actually does not require SIMD instructions in the inner cycle.

The algorithm is based on computing and updating minimum and maximum distances to the traversed cell for all rays in the group. At the beginning of the traversal we will store minimum and maximum distances to the scene's bounding box in the float array **t_cell[2]**. In **Figure 1**, **t_cell[0] = oa** and **t_cell[1] = oD**. For each traversed cell, by using conditions (3) from section 4.1, we can compute two values: **t_plane[0]** – minimum distance to the split plane $\mathbf{P_0}$ in the cell (**oδ**) and **t_plane[1]** – maximum such distance (**oα**). Then

```
// If we can verify that all rays exit the cell
// before intersecting the plane...
if (t_cell[1] < t_plane[0]) {
    // Only the nearest cell is traversed.
    // Compute address of the nearest node and
    // continue traversal. t_cell is not changed.
    ...
}

// If all rays enter the cell
// only after intersecting the plane...
if (t_cell[0] > t_plane[1]) {
    // Only the farthest cell is traversed.
    // Compute address of the farthest node and
    // continue traversal. t_cell is not changed.
    ...
}

// We will have to traverse the nearest cell
// followed by the farthest cell. This branch also
// handles a small percentage of misdiagnosed cases
// (when only one cell is actually traversed).

// The next array will contain t_cell values
// for the farthest cell
float t_farthest[2];
t_farthest[0] = max(t_cell[0], t_plane[0]);
t_farthest[1] = t_cell[1];

// Store t_farthest values and the address
// of the farthest cell into the programming stack
// (it will be used after the subtree starting
// at the nearest node is processed).
...

// Modify the t_cell interval for the nearest cell
t_cell[1] = min(t_cell[1], t_plane[1]);
// Continue traversing nearest cell
...
```

It is evident that this algorithm has the same complexity as one for traversing an individual ray [Wald 2004]. However, it executes traversal of all rays in the group simultaneously by using proxy intervals. Groups cannot be very large as performance will suffer from too many inactive rays in the group (see discussion in section 3.2). In our experiments, we found the best group size to be 4x4, which compares favorably with the 2x2 groups used by Wald.

Even though the interval traversal algorithm sharply reduces the required number of operations compared to the diligent traversal of all 16 rays in the group, overall performance improvement is only about 20%. The reason is that the branches in the kd-tree traversal are data dependent with all 3 continuation scenarios (nearest, farthest or both sub-nodes) occurring with roughly equal probability.

5 Results and Discussion

The simplest way to evaluate the performance impact of a particular feature is to test two implementations, one with and one without the feature. The average performance difference of our MLRTA and non-MLRTA implementations for the 3 scenes in Table 1 is about 3.25X for primary rays and 2.75X for primary and shadow rays. As will be evident from the statistical data in this section, this is roughly equivalent to the reduction in the number of traversal steps executed by the algorithm. Compared with the best results reported elsewhere in the literature, our traditional implementation (without MLRTA) is still about 2X faster. We can only speculate that this is due primarily to different tree construction and somewhat different traversal and intersection methods, API overhead may also play a role. In this section we will provide a more formal quantification of the performance results based on measurements of the mathematical operations.

We will analyze the MLRTA results by providing data for 4 scenes which vary greatly in scene complexity and occlusion properties. For convenience, all results will be presented on a per packet basis - we use packets with 16 rays (4x4). If a total of **k** cells are traversed during the rendering of a 1024x1024 pixel image, the ratio **k/(1024*1024/(4*4))** will be used. We account for all traversal steps regardless of whether they are executed during the EP or XP search. Only those intersection tests which were not avoided through AABB culling are included in the statistics. If a triangle intersection test is avoided then no triangle data is accessed.

Scene / # of triangles and view / Average measurements per 4x4 packet at 1024×1024 resolution		Erw6 (804)	Conference (274K)	Soda Hall (2195K)	Asian Dragon (7M)
number of traversed cells	1. MLRT	3.98	20.87	32.52	32.65
	2. no MLRT	13.00	49.98	71.37	42.18
3. EP search only		0.51	2.30	4.44	2.72
non-masked intersections	4. MLRT	1.09	2.48	1.59	19.97
	5. no MLRT	1.09	2.55	1.52	19.94

Table 2: For primary rays MLRTA significantly reduces the number of traversal steps (first row vs second) without adversely affecting the number of intersection tests.

number of traversed cells	6. MLRT	10.07	53.73	69.07	45.01
	7. no MLRT	24.83	101.06	117.22	58.41
non-masked intersections	8. MLRT	1.25	3.71	2.17	23.51
	9. no MLRT	1.22	3.75	2.09	23.48

Table 3: Corresponding measurements for primary + shadow rays (one light source).

By analyzing rows 1 and 2 of Table 2 we see that the MLRTA greatly reduces the number of traversal steps required. This ratio varies from ~3X for scenes with a lot of occlusion to 1.3X for the last scene which has limited occlusion. The MLRTA's goal is to minimize the number of operations in the most time-consuming part of the ray-tracing pipeline. The return on investment is quite

high. A 10% investment in finding a good EP yields an overall performance improvement of 2.5X (in the conference scene).

By examining rows 4 and 5 we see that there is no significant change in the number of overall intersection tests performed, which is ideally what you would expect (ie finding good EPs helps you avoid redundant traversal of the upper parts of the kd-tree, but has no detrimental effect on the processing at the lower part of the tree). We have observed that the best RT results are achieved when roughly 2/3 of the time is spent traversing the kd-tree and 1/3 actually looking for intersections and that this ideal ratio increases with model size. Because the termination criteria do not depend explicitly on the kd-tree depth, the number of triangles in the leaf nodes remains roughly constant. In most cases individual triangles can show up in multiple leaf nodes. For those leaves some redundant intersection tests can be avoided by using a mailbox mechanism [Amanatides and Woo 1987].

The data in these tables were obtained for the inverse frustum culling algorithm introduced in section 3.3. The direct method requires about 20% more EP traversal steps. Also, a direct frustum culling test is more expensive than an inverse test. Accordingly, the inverse method clearly has an edge, at least for the coherent packages which we are currently using.

We have conducted preliminary tests on using MLRTA to facilitate adaptive geometric anti-aliasing as described above. Preliminary results show that for a given level of quality it results in a 50% reduction in the number of actual rays shot for a given scene (these results were evaluated using static images). In our experience, since we are now able to view most of these scenes at interactive rates on ordinary desktop machines, temporal aliasing artifacts are now more dominant. We are planning to revisit these issues in the future.

The results given in Table 2 were obtained for primary rays. Similar conclusions can be drawn also from analysis of secondary rays. Table 3 includes data for primary and shadow rays for the same 4 scenes (normalized for one primary packet of 4x4 rays). A significant portion of the intersection tests for non-occluded shadow rays can be avoided by excluding objects already hit by the parent primary rays. For this reason the ratio of traversal vs. intersection steps is even higher than for primary rays (compare the quotients of rows 1 to 4 and 6 to 8).

6 Limitations of MLRTA and Future Work

MLRTA does not require advance knowledge of the rays in the group and uses ranges of directions to traverse the whole group at once. Even the interval extension, as described in section 4.3, uses exact rays only during intersection tests and operates with inclusive intervals during traversal. Although this feature facilitates adaptive anti-aliasing of the image, it prevents direct utilization of MLRTA for very “wide” packages with small numbers of rays. In such cases we end up doing a lot of unnecessary speculative work on behalf of rays which will never materialize. This problem cannot be fixed merely by splitting the range data. In **Figure 8**, a big group of secondary rays is represented in some parametric space. If we just split the original voxel uniformly, some sub-voxels will have no rays at all and tracing them would be a waste of time.

If the size of the original group of rays is small compared with number of sub-voxels, it is very unlikely that any sub-voxel will include a large number of rays. In this situation, MLRTA or any other collective traversal mechanism will be ineffective. At the same time, for all secondary rays considered together there exists a partitioning of the parametric space for which there will be substantial amount of sub-voxels with a considerable number of coherent rays in each one. This draws a parallel with the Dirichlet Principle (if you try to place n+1 rabbits into n cells, there will be at least one cell with at least 2 rabbits). We have to select sub-voxels in such a way that they will be large enough to encompass big groups of rays yet small enough to be traversed mostly “together” through the tree.

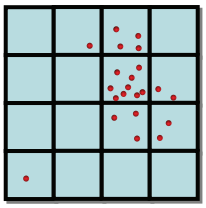


Figure 8: Distribution of secondary rays. Each red dot represents a ray in some parametric space (3D origin + 2D direction). Some voxels have none or very few rays, while others have a lot of coherent rays.

We are planning to research these issues, in particular exploring approaches for culling such 5D voxels first outlined by Arvo and Kirk [1987]. We assume that the 3D component of such a parametric space can be handled implicitly by associating rays with low-level cells in a kd-tree when they are traversed. These cells are usually small as this is one of the goals of kd-tree builder. We can then traverse those voxels with a larger number of constituent rays using the interval approach as described in section 4.3. All possible splits of the directions of the original group can be pre-computed using a simple binning technique to avoid tracing empty groups. Voxels with a small number of rays could be traversed on a per-ray basis.

This is, of course, speculation at this point and whatever approach eventually gets used will have to be compared against tracing individual rays sequentially. Presumably, by selecting the proper size of the original tile and tracing different levels of secondary rays separately (as suggested by Nakamaru and Ohno in [1997]), this could be effective for the majority of scenes.

Considering shadow rays for point lights, they can be handled by MLRTA directly by tracing them from the light sources to the hit points produced by the primary rays. Currently, we implemented a simplified version of this approach by using MLRTA for all the secondary rays which are reflected from flat surfaces (considering reflected and shadow rays). For shadow rays originating from secondary hits, it may be necessary to use partitioning schemes as outlined at the beginning of this section.

MLRTA can certainly also be used in photon mapping (for the final gathering step), area lights, and ambient occlusion schemes [Gritz et al. 2002]. In fact, area lights seem to be well suited for processing using a frustum formed between the hit point and polygonal area lights. We are planning to explore these issues in the near future.

7 Summary

MLRTA uses geometric properties of a large group of rays to find a common entry point into the kd-tree for all of the rays in the group, thus avoiding redundant operations. This approach enables us to find correct intersection points by using just 1/3 of the traversal steps which would otherwise be required.

The entry point search is carried out by identifying common group properties and using these properties in lieu of rays. We

analyzed 2 different ways of defining such group properties. In one, a set of planes enclosing all the rays is created and traversed through the kd-tree using the direct frustum culling algorithm. This approach works well in traditional CG applications where the frustum is 'big' and objects are typically 'small' and can be effectively culled against the frustum by using the frustum's planes. For ray-tracing applications however, the opposite characterization is more likely. It allows us to "invert" the traditional frustum culling algorithm, that is to cull the frustum by using the faces of the AABBs. This new inverse frustum culling algorithm is broader in scope and does not include the notion of frustum bounding planes. Accordingly, it can be used for more general collections of coherent rays.

Another attractive property of the MLRTA algorithm is that it provides a natural measure of the geometric complexity of specific view directions. We intend to continue investigating these issues, paying particular attention to anti-aliasing in the temporal domain [Martin et al. 2002]. An appealing approach would be to track groups of rays through multiple time frames.

Acknowledgments

Many thanks to Ingo Wald, Philipp Slusallek, and Carsten Benthin for sharing their results and models. The Asian Dragon and Thai Statuette models are courtesy of the Stanford 3D Scanning Repository and room model from the accompanying video is courtesy of the Cornell University Graphics Group. The authors would like to thank the anonymous reviewers for their valuable comments and pointing out to missing references. We gratefully acknowledge discussions with and assistance from Radek Grzeszczuk and Gordon Stoll.

References

AMANATIDES, J. 1984. Ray Tracing with Cones, In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18, 4, ACM, 129-135.

AMANATIDES, J. and WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. *Eurographics Conference Proceedings 1987*, 3–10.

ARVO, J. and KIRK, D. 1987. Fast Ray Tracing by Ray Classification, In *Computer Graphics (Proceedings of ACM SIGGRAPH* 87), 21, 4, ACM, 55-64.

ASSARSSON, U. and MÖLLER, T. 2000. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools*, 5, 9-22.

BENTHIN, C., WALD, I., and SLUSALLEK, P. 2003. A Scalable Approach to Interactive Global Illumination, *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22(3), 621-630.

CHO, F.S. and FORSYTH, D. 1999. Interactive ray tracing with the visibility complex. *Computers and Graphics (Special Issue on Visibility - Techniques and Applications)*, 23(5), 703-717.

DAVIS, T. and DAVIS, E. 1999. Exploiting frame coherence with the temporal depth buffer in a distributed computing environment, *Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, 29-38.

DMITRIEV, K., HAVRAN, V., and SEIDEL, H.-P. 2004. Faster Ray Tracing with SIMD Shaft Culling, *Research Report, Max-Planck Institut Für Informatik*, MPI–I–2004–4–006.

GENETTI, J., GORDON, D., and WILLIAMS, G. 1998. Adaptive Supersampling in Object Space Using Pyramidal Rays. *Computer Graphics Forum*, 16(1), 29-54.

GHAZANFARPOUR, D. and HASENFRATZ, J-M. 1998. A Beam Tracing with Precise Antialiasing for Polyhedral Scenes. *Computer & Graphics*, 22(1), 103-115.

GLASSNER, A. 1984. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics & Applications*, 4(10), 15-22.

GRITZ, L., APODACA, T., QUARONI, G., BREDOW, R., GOLDMAN, D., LANDIS, H., and PHARR, M. 2002. RenderMan in Production. *ACM SIGGRAPH 2002 Course Notes*, Course 16.

HAVRAN, V. and BITTNER, J. 2000. LCTS: Ray Shooting using Longest Common Traversal Sequences. *Computer Graphics Forum*, 19(3), C59-C70.

HAVRAN, V. 2000. *Heuristic Ray Shooting Algorithms*, Ph.D. Thesis, Czech Technical University.

HAVRAN, V., BITTNER, J., and SEIDEL, H.-P. 2003. Rendering: Exploiting temporal coherence in ray casted walkthroughs, *Proceedings of the 19th Spring Conference on Computer Graphics (SCCG 2003)*, 149-155.

HECKBERT, P. and HANRAHAN, P. 1984. Beam tracing polygonal objects. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18, 4, ACM, 119-127.

KAY, T. L. and KAJIYA, J. T. 1986. Ray Tracing Complex Scenes, In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, 20, 4, 269-278.

MACDONALD, J. and BOOTH, K. 1990. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6, 153-166.

MARTIN, W., REINHARD, E., SHIRLEY, P., PARKER, S., and THOMPSON, W. 2002. Temporally Coherent Interactive Ray Tracing. *Journal of Graphics Tools*, 7(2), 41-48.

NAKAMARU, K. and OHNO, Y. 1997. Breadth-First Ray Tracing Utilizing Uniform Spatial Subdivision, *IEEE Transactions On Visualization and Computer Graphics*, 3(4), 316-328.

OHTA, M. and MAEKAWA, M. 1990. Ray-bound tracing for perfect and efficient anti-aliasing. *The Visual Computer: International Journal of Computer Graphic*, 6(3), 125-133.

RAMASUBRAMANIAN, M., PATTANAIK, S., and GREENBERG, D. 1999. A perceptually based physical error metric for realistic image synthesis. In *Proceedings of ACM SIGGRAPH 1999*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 73-82.

SHINYA, M., TAKAHASHI, T., and NAITO, S. 1987. Principles and applications of pencil tracing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21, 4, ACM, 45-54.

SZIRMAY-KALOS, L., HAVRAN, V., BALAZS, B., and SZÉCSI, L. 2002. On the Efficiency of Ray-shooting Acceleration Schemes. *Proceedings of the 18th Spring Conference on Computer Graphics (SCCG 2002)*, 89-98.

TELLER, S. and ALEX, J. 1998. Frustum Casting for Progressive, Interactive Rendering. *Technical Report, Laboratory for Computer Science, Massachusetts Institute of Technology*, TR-740.

WALD, I., SCHMITTLER, J., BENTHIN, C., SLUSALLEK, P., and PURCELL, T.J. 2003. Realtime Ray Tracing and its use for Interactive Global Illumination, STAR, *Computer Graphics Forum (Proceedings of Eurographics 2002)*, 22(3).

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

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