Accelerating Algorithms for Ray Tracing

Article · December 2006

CITATIONS
0

READS
2,791

1 author:

Hector Antonio Villa-Martinez
Universidad de Sonora (Unison)
21 PUBLICATIONS **34** CITATIONS

SEE PROFILE

Advanced Algorithms – Fall 2006: Accelerating algorithms for Ray Tracing

Héctor Antonio Villa Martínez
havillam@mtu.edu
Michigan Technological University
Department of Computer Science

December 2006

1 Introduction

Ray Tracing is one of the main rendering methods used to obtain photo-realistic synthetic 3D images. When used with other techniques, like texture mapping and photon mapping, it can generate very realistic images, see Figure 1. Ray Tracing is popular not only because it can render nice images, but because it is easy to implement. The problem is that the basic algorithm is very time consuming and it can only be used for the simplest scenes.

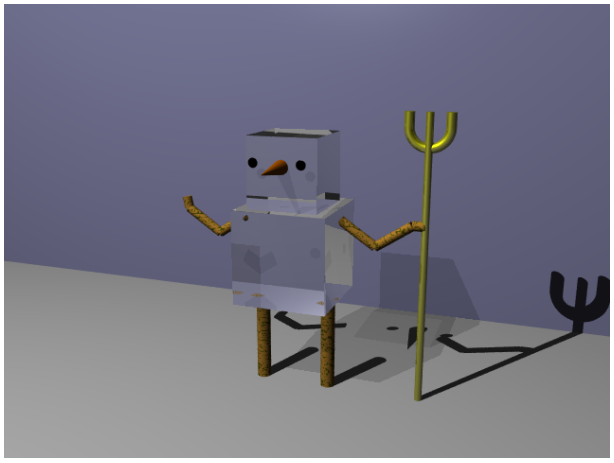


Figure 1: Figure modeled and rendered using Pov-Ray.

The objective of this report is twofold. First, to explain why the "naïve" Ray Tracing algorithm is expensive. Second, to present a survey of the main techniques used to reduce the complexity of the basic algorithm.

The report is organized as follows. In Section 2, there is a description and a complexity analysis of the basic Ray Tracing algorithm. Section 3 presents a summary of a classification of efficient

techniques found in Watt book [1, p. 354-367]. Here we learn that spatial coherence methods are the only ones that reduce the complexity of the naïve ray tracing algorithm. Then, in Sections 4 to 7, we review 7 articles which describe some of these coherence techniques: object and ray coherence, octrees, grids, and BSP and kd-trees. Finally, Section 8 has a short summary of parallel ray tracing, and Section 9 has the conclusions.

2 Ray Tracing

Ray Tracing simulates the path of a light ray into a scene composed of various geometric bodies. The simulation takes in account that a ray can be absorbed by a diffuse body, reflected if the body is reflective, or refracted in case the body is transparent or traslucid.

In order to achieve this simulation, a ray tracer places a display screen between the viewer and the scene (see Figure 2). For each pixel, the renderer shoots a ray from the viewer eye, through the pixel, into the scene. Then, the intersection of the ray with the first scene element in the ray's path is computed. At this intersection point, the ray is divided in one or two secondary rays depending on the object material: a reflection ray and a refraction ray. These rays are traced recursively, computing at each step the intersection point with objects, until a predetermined depth is reached, or the ray miss all objects and goes outside the scene.

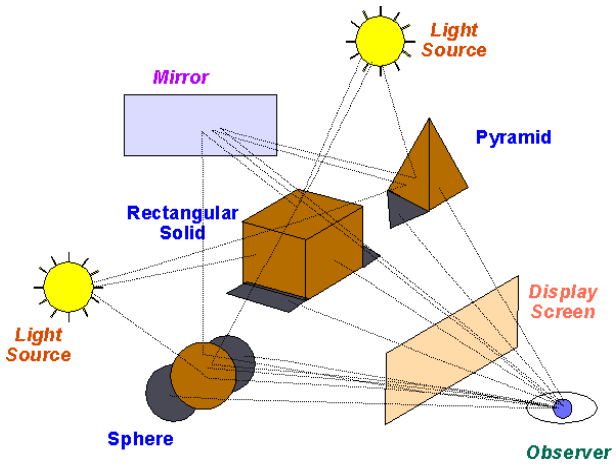


Figure 2: Basic ray tracing algorithm.

Consider the pseudocode for a ray tracer:

```
ray_tracer
// The display screen resolution is w x h
  for x := 1 to w
    for y := 1 to h
      maxdepth := d
      Make a ray from the camera to the pixel <x, y>
      shoot_ray (ray, maxdepth) // Shoots a ray into the scene
```

The time complexity of the basic algorithm is $O(whr)$, where w and h are the width and the height

of the display screen in pixels; and r is the complexity of shooting a ray into the scene.

Now, consider the pseudocode for shooting a ray into the scene (based in the pseudocode presented by Watt [1, p. 349]):

```
shoot_ray (ray, depth)
  check if the ray intersects an object
  if there is an intersection
    compute local color intensity
    depth := depth - 1
    if depth > 0
      compute the reflected ray 'l'
      compute the refracted ray 'r'
      return local color + shoot_ray (l, depth) + shoot_ray (r,depth)
    else // depth limit
      return local color
  else  // no interception
      return background color
```

The best case for shooting a ray is $O(1)$, when the original ray, the ray shot from the camera to the scene, hits a diffuse and opaque body (there is no reflected nor refracted rays), or if the ray misses all the objects. The worst case is $O(d)$, where d is the maximum level of recursion established either by the programmer or the user.

However, the most expensive part of shooting a ray is the first line, checking if the ray intersects an object, because the algorithm needs the first intersection point, that is, the closest intersection to the ray origin. Then, in a brute force approach, the algorithm intersects the ray with every object in the scene, and potentially, with all polygons that compose a complex object, and returns the closest intersection point. This procedure is so expensive, that in fact, a naïve ray tracer can expend almost 100% of its running time finding intersections [2].

From the above analysis, we can conclude that a more realistic worst case time complexity for shooting a ray is $O(n\,d)$, where n is the number of objects (or alternatively, the number of polygons) in the scene. Finally, since d is a constant, the time complexity depends only on n, and it is $O(n)$.

Thus, the time complexity for the naïve Ray Tracing algorithm is $O(w\,h\,n)$. This complexity is not surprising. It means, that given a fixed display screen resolution of w x h pixels, a scene with a large number of objects will be more time consuming to render than a scene with less objects.

Then, all the approaches aimed to reduce the complexity of the brute force Ray Tracing algorithm, are focused in reduce the dependence on the number of objects in the scene. The ideal being to obtain a time complexity of $O(w\,h)$, that is, an algorithm that only depends on the display screen resolution, and where it takes roughly the same time to render a complex scene than a simpler scene.

3 Classification of accelerating schemes

Watt[p. 354-367] [1] presents a useful classification about the main lines of investigation in accelerating the basic Ray Tracing algorithm:

- Adaptive depth control. Here, the idea is to attack the hidden constant d, the maximum recursion depth, by only shooting a ray, either a reflective or a refractive ray, if the contribution of the ray will be above a predefined threshold. A reflective ray will be attenuated if it hits a highly diffuse object. In the same way, a refractive ray will almost have not contribution to the final color if it hits an opaque object. Then, depending on the threshold, a ray tracer can decide not to recurse further. Watt[p. 355] [1] cites a study that reports an average depth of 1.71 in a highly reflective scene. However, even with this reduction in rendering time, the theoretical worst case time complexity remains $O(w\,h\,n)$.

- First hit speed up. Here, the cost of finding the first intersection is transferred to a preprocessing step. Then, at running time, each ray can retrieve a pointer to its first intersection from a data structure. The main problem with this approach is that it requires a massive amount of storage. Furthermore, it only reduces the complexity of the first intersection, the following intersections still need to be computed.

- Bounding objects. It is well known that finding the intersection of a ray with a sphere or a box, is much easier than finding the intersection with a more complex figure [1] [3] [4]. For example, Haines presents an efficient ray-sphere intersection algorithm that requires only a total of 16 additions/subtractions, 13 multiplies, 1 square root, and 3 comparisons [5, p. 43]; and a ray-box intersection algorithm that requires 2 subtractions, 2 divisions, and 5 comparisons [5, p. 65]. Then, in the bounding objects approach, the objects in the scene are bounded by spheres or boxes, calling them the object's *extent*. There is a reduction in the time spent doing intersections tests, because the expensive intersection of the ray and a complex object is only computed, if the ray intersects the object's extent. Yet, the time complexity of the worst case is still $O(w\,h\,n)$, because there are n extents, one for each object.

- Bounding hierarchies. This approach is an extension of the previous one, and tries to exploit the fact that in most cases, the objects in a scene are not uniformly distributed. Instead of enclosing each object in its own extent, a group of objects, sufficiently close to each other, are bounded by a single extent. Bounding hierarchies reduces the time complexity from $O(w\,h\,n)$ to $O(w\,h\,m)$, where $m < n$ is the number of extents. Watt says that the main disadvantages of this approach are its dependency on the nature of the scene, and the investment required to find a suitable hierarchy [1, p. 358].

- Spatial coherence. In this approach, the main idea is divide the space occupied by the scene in regions, and then, given a ray, only the objects that are in the ray's region are tested for an intersection. The space subdivision is done in a preprocessing step, storing the information about the space occupancy in an auxiliary data structure. The most common data structures used in this approach are: BSP (binary space partitioning) trees, quadtrees, octrees, and k-d trees. At rendering time, the data structure is traversed to retrieve the objects that are in the same region where the ray is, bringing the complexity down to $O(w\,h\log n)$. Watt concludes that this approach is the most promising in reducing the complexity of Ray Tracing, and cites some early (pre 1990) work in this area [1, p. 358-363].

There are two other approaches not discussed by Watt: parallel Ray Tracing and hardware Ray Tracing. Hardware Ray Tracing is not discussed either in this report. section 8 has a short summary in parallel Ray Tracing.

Watt prediction about spatial coherence has been proved true. Most of the recent research in accelerating Ray Tracing is in developing better spatial coherence algorithms. Thus, in the rest of this report, we will review some of the latest work done in that area.

4 Object and Ray Coherence

The Merriam-Webster OnLine dictionary (`http://www.m-w.com`) defines coherence as a "systematic or logical connection or consistency". Gröller and Purgathofer [6] surveyed the use of coherence in Computer Graphics, and says that coherence is useful because in many situations, scene properties do not change abruptly but in a smoothly way. They list 10 basic types of coherence in Computer Graphics. Two of them, object and ray coherence, are used by González and Gisbert [7] to reduce the number of intersections and the running time of a ray tracer.

Object coherence is based in relationships between objects or between parts of the same object. Sometimes, the objects are clustered and there are big empty spaces. From a voxel point of view, it is very possible that its neighbors are occupied by the same object [6, 7].

Ray coherence means that it is very possible that similar rays (i.e. rays with similar origin and direction) will intersect the same object, and that the intersection points will be close to each other [6, 7].

4.1 Implementation

González and Gisbert implemented a *beam* tracer instead of a ray tracer. A beam can be seen as a pyramid of arbitrary base length. Equivalently, a beam can also be seen as a set of rays with the same origin and which crosses some arbitrary planar polygon (see Figure 3).

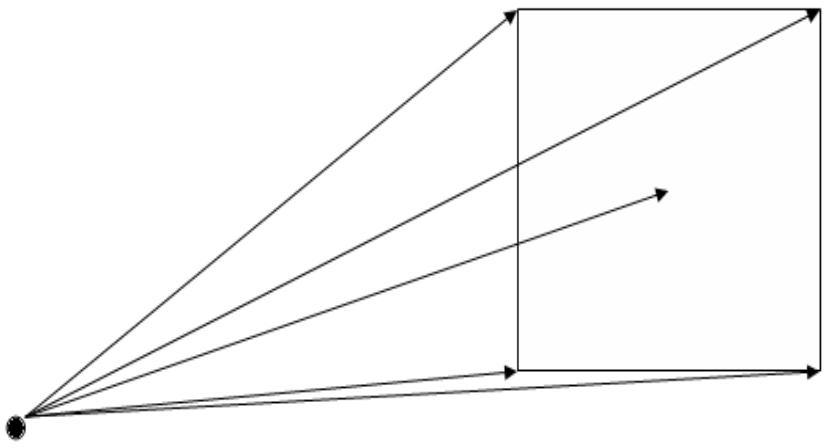


Figure 3: A beam with five rays.

The beam tracer uses a regular octree for scene management. The main difference between using beams and using rays, is the way a beam traverses the octree. When a beam arrives to a voxel[1] there are two possibilities:

- The voxel is empty. The beam tracer determines the number of neighbors to this voxel. If there are none, the beam has exited the scene and the tracing for this beam finishes. If there is one neighbor, the beam passes intact to the neighbor. If there are two or more neighbors, the beam is divided in function of the neighbors sizes. In the last two cases, the tracing continues recursively.
- The voxel is occupied. In this case, each ray in the beam is intersected with the objects in the voxel. If the number of intersections are above a threshold, each ray is traced individually. Otherwise, tracing is continued with the beam, but only with the rays that did not have an intersection.

4.2 Results

González and Gisbert compared their beam tracer with two ray tracers that also uses octrees, and with a ray tracer based in bounding volumes. For the comparison they used two scenes, one with 400 randomly distributed spheres, the other with 91 highly concentrated objects.

For the first scene, the bounding volumes ray tracer behaved badly. This is understandable, because bounding volumes works well when objects are not uniform distributed. Of the three octree-based tracers, the beam tracer was slightly better that the two ray tracers.

For the second scene, the bounding volumes ray tracer and the beam tracer performed much better that the two ray tracers. The reason is that because the scene has big empty areas, the beam tracer does not need to trace rays individually.

The authors conclude that their implementation presents the best of the two worlds, because beam tracing takes advantage of object and ray coherence, and performs well, both in uniform and not uniform distributed scenes.

5 Improved Octrees

An octree is a data structure used to represent a 3-D space, recursively dividing the space in blocks *voxels*, until a condition is satisfied [8, p. 211]. Figure 4 shows how the space partition is represented as an octree. Note that the division does not need to be uniform, and that not necessarily each node must have eight children, because the partition depends on the nature of the application.

After the octree has been built, the leaves contain the list of objects that insersect each voxel. The internal nodes provide only the paths to the leaves. The benefit of using an octree to represent a scene is that, instead of testing a ray against all the scene objects, a ray tracer only examines the objects that intersect with the voxel that contains the ray. If there is no intersection in the current voxel, the ray tracer finds the next voxel where the ray will be, and repeats the intersection testing [9].

[1] Remember that voxels are stored in the octree leaves.

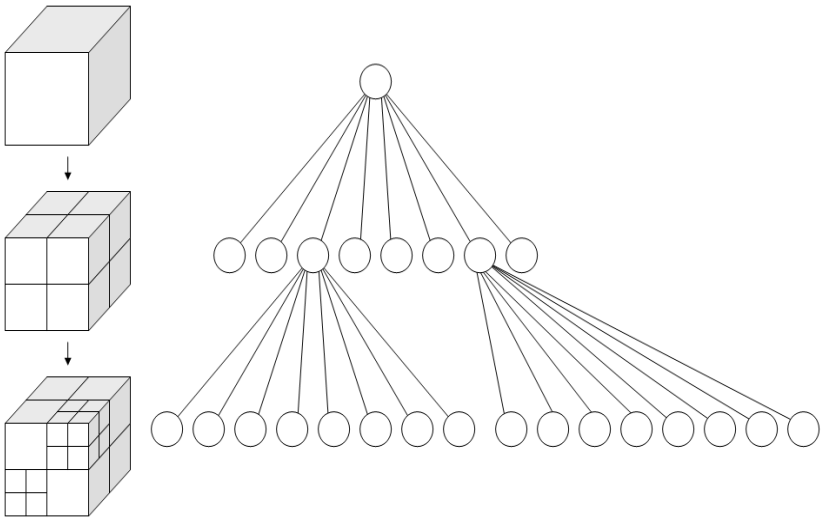


Figure 4: An octree, depicted as dividing a 3-D space. Copied from Wikipedia, http://de.wikipedia.org/wiki/Bild:Octree.png, under the terms of the GNU Free Documentation License, Version 1.2.

The main disadvantage of octrees is that they have the problem of the "teapot in a stadium" [10], where the representation of a small object, inside a big empty scene, can produce a depth octree and a waste of space in empty voxels.

5.1 Octree-R

The Octree-R, first proposed by Whang et al. [11], is a variant of the traditional octree. The difference is in the criteria used to divide the space. Using a probabilistic model, the Octree-R construction algorithm finds an estimation of the number of ray-object intersection tests, and then, splits the space using the plane that minimizes that estimation.

5.1.1 Construction of an Octree-R

Suppose there are two convex volumes, A and B, and that A contains B (see Figure 5). Whang et al. says that a good estimation for the conditional probability for a ray r to intersect B once r has entered A is:

$$Pr(B|A) = \frac{S_B}{S_A} \tag{1}$$

where S_B and S_A are the surface area of B and A respectively.

Now, the authors attack the general case, where a convex space R is divided in two spaces A and B (see Figure refvol2). Using equation 1, the authors determine the expected number of ray-object intersection tests, $E(t)$, when a ray crosses space R is:

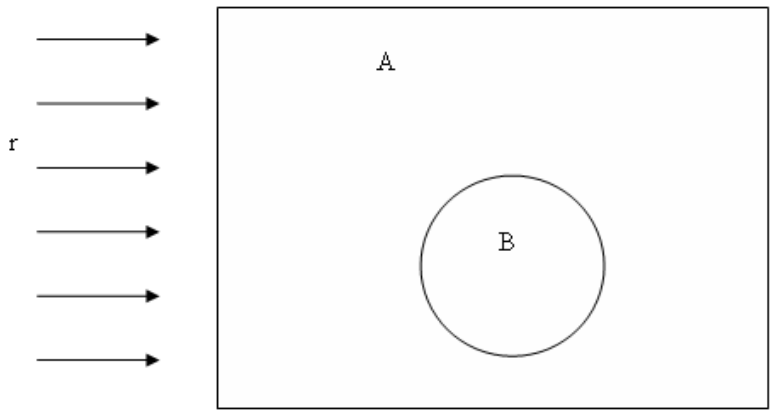


Figure 5: Computing the conditional probability for a ray r to intersect B once it has entered A. Based in Fig. 1 in [11].

$$E(t) = \frac{t(b+c)+bc}{a(b+c)+bc}n(t) + \frac{(a-t)(b+c)+bc}{a(b+c)+bc}m(t) + \frac{a(b+c)+2bc}{a(b+c)+bc}s(t) \tag{2}$$

where:

- $n(t)$ is the number of objects completely included in A.
- $m(t)$ is the number of objects completely included in B.
- $s(t)$ is the number of objects intersecting with plane t.

The algorithm for building an Octree-R is the same that for a regular octree. The only exception is the computation of the t value that minimizes equation 2. Another feature, is that in order to avoid inspecting all the space, the authors cite a paper where it was found that the optimal splitting plane lies between the spatial median and the object median. The plane is found sampling ten positions between the spatial and the object median.

5.1.2 Results

Whang et al. compared their implementation of Octrees-R against an implementation using regular octrees. They used four object data sets with 3000, 4000, and 5000 voxels, and two distributions (uniform and gaussian). When the objects are uniform distributed the benefit of using Octrees-R is small. There is a reduction in run time from 4% for 3000 voxels to 12% for 5000 voxels. The authors argue that the reason is because the optimal splitting plane t is near the object median, and the space is subdivided in voxels of roughly the same size, like in a regular octree. When the objects are not uniform distributed, the reduction in run time is from 9% (3000 voxels) to 47% (5000 voxels).

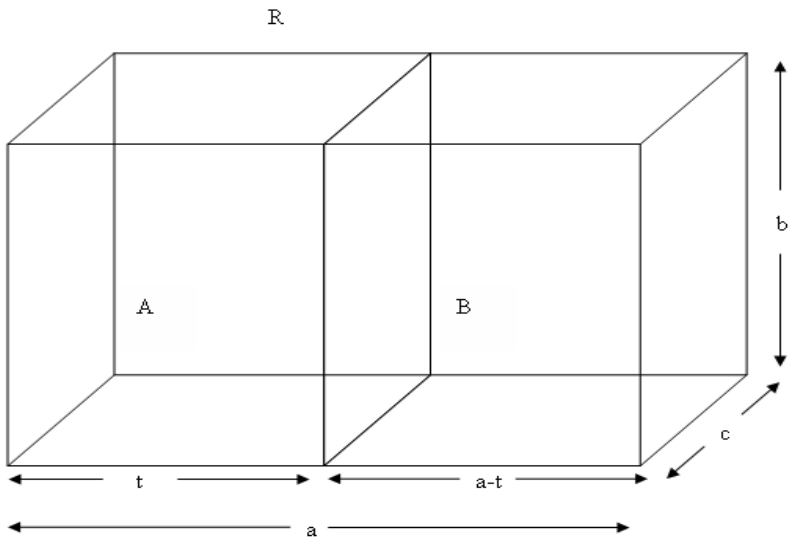


Figure 6: The general case where space R is divided in A and B by the plane t. Based in Fig. 2 in [11].

There is time penalty in the Octrees-R construction algorithm. From one data set showed by the authors, building an Octree-R costs 7-8 times the time of building a regular octree. Whang et al. says this is acceptable since this time is less than 5% of the total rendering time.

5.2 Cost-driven octree construction

Aronov et al. [2] introduced a cost predictor for evaluate the performance of a ray-shooting data structure on a data set. While used in the construction of an octree, the cost predictor provides a degree of lookahead. If the subdivision of a voxel does not lead to an improved cost, the voxel subdivision is stopped. Using the cost predictor, the authors evaluated several octree construction schemes.

5.2.1 The cost predictor

The cost predictor was introduced in a previous paper by the same authors [12]. Given a decomposition, $\mathcal{T}$, of a bounding box of the scene into simple convex voxels, the efficiency of $\mathcal{T}$ is measured by $E(\mathcal{T})$ as follows:

$$E(\mathcal{T}) = \frac{W(\mathcal{T})}{A(\mathcal{B}) + \sum_{s \in S} A(s)} \tag{3}$$

where $W(\mathcal{T})$, the simplified weighted work performed during a traversal, is defined as:

$$W(\mathcal{T}) = \sum_{\mathcal{B}_i} (1 + |S_i|) \times A(\mathcal{B}_i)$$

and $A(\mathcal{B})$ is the surface area of the outermost bounding box, $A(s)$ is the surface area of object s, $\mathcal{B}_i$ is a voxel in the decomposition, and S_i is the set of scene objects meeting $\mathcal{B}_i$.

Finally, if the cost, α, of computing a ray-object intersection is different from the cost, β, of computing a ray-box intersection, W can be refined to:

$$W(\mathcal{T})_{\alpha,\beta} = \sum_{\mathcal{B}_i} (\alpha + \beta |S_i|) \times A(\mathcal{B}_i) \tag{4}$$

Despite some simplifications done in the derivation of E and W, that left some rays unaccounted, Aronov et al. claim that E is a sound measure for the quality of the decomposition.

5.2.2 Implementation

The octree implementation described by Aronov et al. can adjust its construction criteria and produce different octrees. As in a regular octree, a voxel which do not meet a *termination condition* is recursively subdivided. The novelty is that the implementation described in this article can be configured and supports a choice of termination conditions. The termination conditions consists of two classes:

1. Separation termination conditions.
   (a) Maximum octree depth allowed.
   (b) Maximum number of objects in a leaf. This criteria is not enforced if the maximum depth has been reached.
2. Cost-driven greedy criteria.
   (a) Greedy without lookahead or 1-greedy. The voxel is subdivided in eight voxels, if the cost given by equation 3 is reduced by the division.
   (b) Greedy with lookahead or k-greedy ($k > 1$). The algorithm evaluates the smallest cost of a subtree with depth at most k and root in the current voxel. If the cost is reduced, the voxel is subdivided.
   (c) Substantial improvement (with or without lookahead). The voxel is divided if the cost decreases by at least of $t\%$, where t is fixed constant.

The ray traversal is performed as a normal octree. The ray tracer locates the origin of the ray descending from the root to the leaves, and then traverses from leaf to leaf testing for ray-object intersections.

5.2.3 Results

Aronov et al. tested their octree implementation using scenes from the ACM Standard Procedural Databases (`http://www1.acm.org/pubs/tog/resources/SPD/`), and from the Stanford 3D Scanning Repository (`http://www-graphics.stanford.edu/data/3Dscanrep/`). In particular,

the authors were interested in four variables, the first two associated with the octree construction, and the other two with the run-time processing:

- Estimated cost using the cost predictor from equation 3.
- Octree size, defined as the sum of the total number of external and internal nodes in the octree, and the total number of objects in the leaves.
- Actual cost, defined as the total number of intersections divided by the number of accesses to the octree.
- Runtime cost, that is the CPU time.

The scheme that showed the best performance for all the scenes in octree total size, construction time, and ray-tracing time, was the 1-greedy strategy with substantial improvement ($t = 50$) and large α/β ratio.

Another interesant result is that 1-greedy strategy has a comparable cost with the traditional heuristic called $j2$[2]. The difference is that 1-greedy produces octrees that are 4/5 times smaller that those produced by $j2$.

6 Improved grids

A regular grid is a space subdivision technique were all voxels have the same size [13] (see Figure 7).

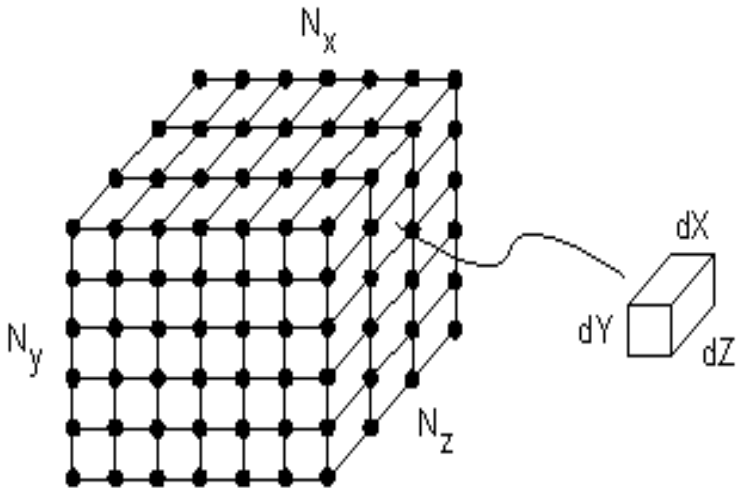


Figure 7: A regular grid. The space is subdivided in voxels with the same size.

Regular grids have the disadvantage of being inefficient in sparse scenes [10], but they are very used in animation, because grids are faster to rebuild than an octree or kd-tree [14].

[2] $j2$ strategy subdivides a voxel until it contains at most 2 objects.

6.1 Adaptive grids

In order to solve the waste of space, that regular grids and octrees have when representing sparse scenes, Klimaszewski and Sederberg [10] proposed a technique called "adaptive grids". The main advantage of adaptive grids over regular grids and octrees, is its efficiency on empty spaces (see Figure 8). The authors claim that in scenes with highly irregular object distributions, adaptive grids outperform regular grids by two orders of magnitude.

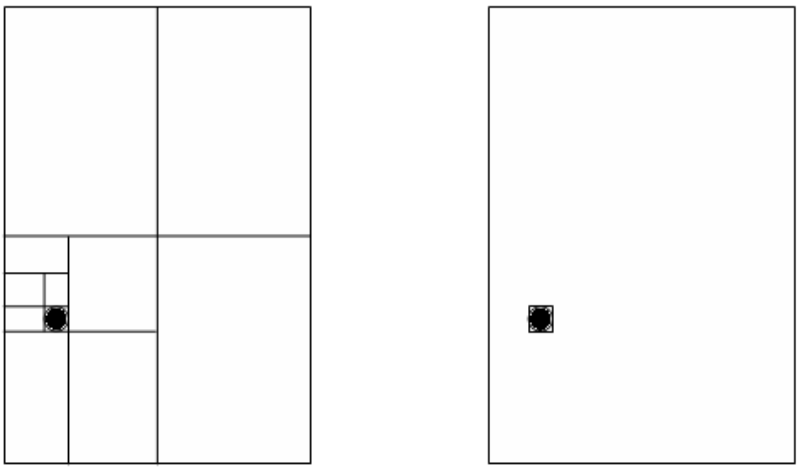


Figure 8: A sparse scene. Left: The octree will have big depth and most of its voxels will be empty. Right: An adaptive grid fits better the lonely object. Based on Figure 2 in [10].

6.2 Implementation

The algorithm to build an adaptive grid starts, either with a regular grid, or with a set of bounding volumes over objects or clusters of objects. Then, the algorithm merge close grids, insert the surviving grids in a tree, and merge grids which are too big or are underpopulated. In a final step, the algorithm voxelizes (divides) the grids.

There are three cases why a grid can be replaced or merged with another grid:

1. There are two overlapping grids with areas A_1 and A_2. If the two grids can be merged into a grid with area A, and $A/(A_1 + A_2) < f$, where f is an arbitrary factor, the merge is performed. Otherwise, the two grids survive. See Figure 9.

2. There is one grid with area A_1, and it is surrounded by another grid with area A. If $A_1/A > m$, where m is an arbitary factor, grid A_1 is merged into A. See Figure 10.

3. There is an underpopulated grid (according to some criteria). The grid is deleted and replaced by the bounding boxes of the objects that were in the grid (see Figure 11). These bounding boxes can be merged later by any of the two first cases.

The algorithm to build an adaptive grid can be summarized as follows:

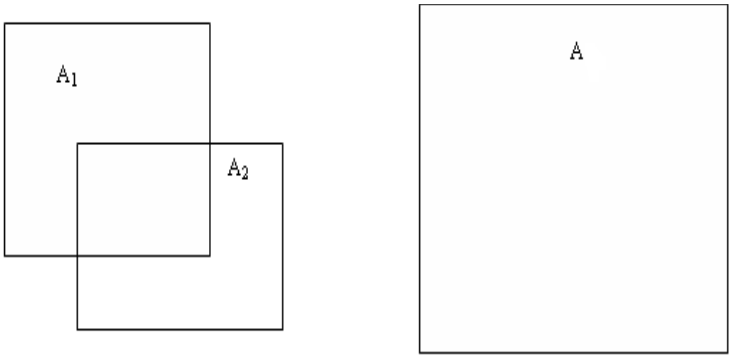


Figure 9: Left: Two candidate grids to be merged. Right: If $A/(A_1 + A_2) < f$, with f an arbitrary factor, A_1 and A_2 are replaced by A. Based on Figure 4 in [10].

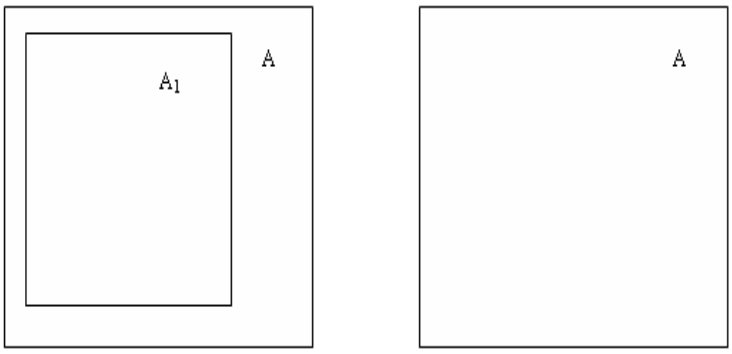


Figure 10: Left: Two candidate grids to be merged. Right: If $A_1/A > m$, with m an arbitrary factor, A_1 is merged into A. Based on Figure 4 in [10].

```
Build-Adaptive-Grid
  for all original nodes
    surround node with a bounding box

  for all bounding boxes
    merge close boxes (merge case 1)

  for all surviving boxes
    insert into a tree using minimum surface criteria
    if a box is too large (merge case 2)
      merge box with its parent
    if a box is underpopulated (merge case 3)
      replace with objects bounding boxes

  for all surviving boxes
    voxelize box
```

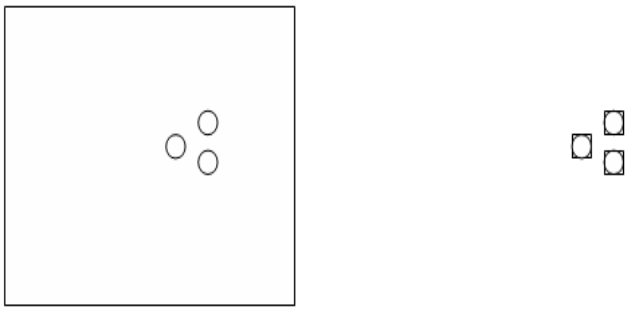


Figure 11: An underpopulated grid (left) is replaced by the objects bounding boxes (right). Based on Figure 5 in [10].

In the last step of the algorithm, the remaining grids are voxelized. If the number of objects in one grid is above a threshold, the grid is divided. These subvoxel grids are heterogeneus, because the number of voxels is proportional to the lengths of the original grid edges:

$$N_3 = \left\lceil \sqrt[3]{\frac{nx_3^2}{x_1 x_2}} \right\rceil, N_2 = \left\lceil \sqrt{\frac{nx_2}{N_3 x_1}} \right\rceil, N_1 = \left\lceil \frac{n}{N_2 N_3} \right\rceil$$

where n is the number of objects in the grid, N_1, $N2$, and N_3 are the number of voxels in the edges with lenghts x_1, x_2, and x_3, respectively. And the brackets mean a round-up operation.

The complexity of the construction algorithm is $O(n)$, where n is the number of original nodes. A node can be a simple object or a polygon of a complex one. The authors argue that the overall cost is not too high because the number of boxes is reduced significantly after the merging of close grids.

6.3 Results

Klimaszewski and Sederberg compared an implementation of their method against a ray tracer using a regular grids approach. They used eight scenes ranging from sparse to compact. In all the eight scenes, the adaptive grids ray tracer outperformed the regular grids ray tracer. In the most sparse scene, the speedup was of 671, while in the most compact scene was of just 5.1. The adaptive grids approach generates the double of voxels than the regular grids technique. The preprocessing time is also the double for adaptive grids.

Finally, the authors compared their algorithm with and without subvoxel grids. The use of subvoxel grids accelerated the rendering time up to 50%. They do not report if there are memory space or construction time penalties.

In their conclusion, Klimaszewski and Sederberg write that they agree with previous studies conclusions that a space partitioning, which presents the best rendering time for one scene, often fails with other scenes. And that maybe the answer is in specialization.

6.4 Coherent grid traversal

Coherent grid traversal, recently proposed by Wald et al. [14], is a technique that enables the use of *coherent rays* in a regular grid-based ray tracer. Coherent rays are rays with the same origin and similar direction. Thus, they can be traced in packets at the same time, reducing the computation time. Packet tracing can be easily implemented if the ray tracer uses kd-trees, but it is not applicable in regular grids. See Figure 12.

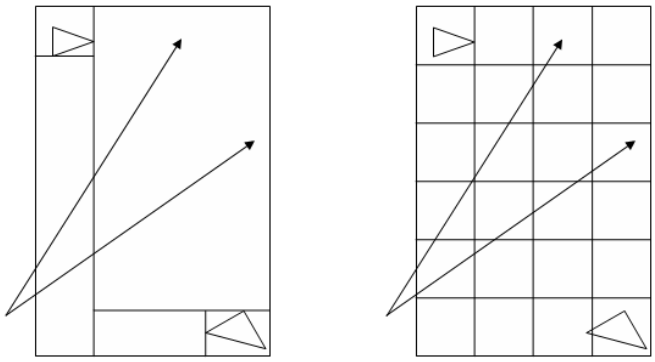


Figure 12: Packet tracing. Left: In a kd-tree, the space adapts to the geometry. Then, the packet of rays travels in the same voxel. Right: In a regular grid, each voxel has fixed size and each ray can visit a different voxel, making packet tracing impractical. Based on Figures 2 and 4 in [14].

Using coherent rays, a kd-tree-based ray tracer outperforms a grid-based ray tracer [15]. However, the main disadvantage of kd-trees is that they cannot be used in dynamic scenes, where at each frame objects move, because k-d trees are very costly to construct (about $O(n \log n)$). On the other hand, a grid can be created and modified at interactive rates [16]. As a consequence, regular grids are attracive for animation.

By implementing packet tracing in regular grids, Wald et al. [14] claims that their ray tracer can compete with any packet-based kd-tree ray tracer.

6.4.1 Implementation

The packet traversal algorithm is straightforward. Instead of one ray visiting cell by cell, a packet of rays visit slice by slice[3]. The algorithm computes the packet *frustum*[4], and found those cells that form the intersection between the frustum and the slice. See Figure 13.

Once the overlapping cells has been determined, all the rays in the packet are intersected with all the objects in each overlapped cell. Since objects can occupy more than one cell, it is possible to have repeated intersections. In order to avoid this repetitions, objects can be tagged with the packet ID. Then, if an object has been tagged, it is skipped from further intersections tests with a ray from the packet with that ID. Another possible source of inefficiency is when an objects is inside the overlapping cells, but outside the frustum. For this case, an frustum culling algorithm is applied.

[3] In this context, a slice is equivalent to a column of cells.

[4] Basically, the packet frustum is the maximum and minimum value of a ray coordinate in each dimension.

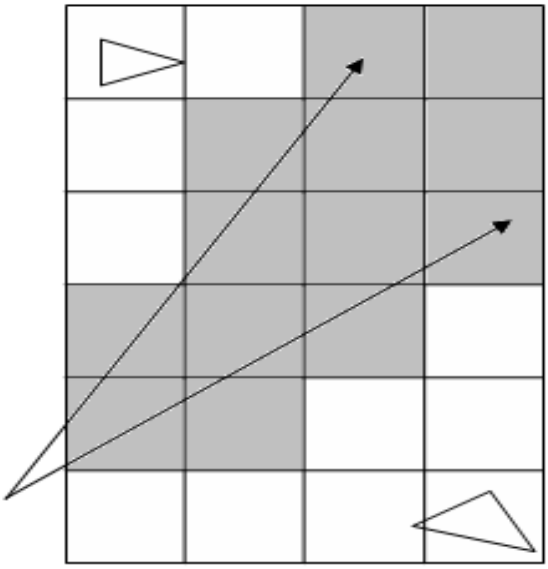


Figure 13: The packet frustum are computed slice by slice from left to right. The grey are is the intersection between the frustum and the slice, i.e. each cell that is touched by a ray in the packet. Based on Figure 3 in [14].

For animated scenes, the grid is built from scratch for every frame. The grid resolution is a function of the number N of objects:

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

where $\overrightarrow{d} = \langle d_x, d_y, d_z \rangle$ is the diagonal, V is the volume of the grid, and λ is a parameter for which the authors determined experimentally that 5 if the best value.

6.4.2 Results

Wald et al. did a series of tests with their implementation. Among the main results they report are:

- Frustum culling and object tagging reduces the number of ray-object intersections by a factor from 8.5 to 14, making coherent grid traversal competitive with kd-tree implementations in this aspect.
- For static scenes, the system is competitive in running time with OpenRT (`http://www.openrt.de/`) and MLRT [17].
- The system is 6 to 21 times faster that a grid-based ray tracer without coherent grid traversal.
- For small and medium-size animated scenes, the rebuilding is fast. Wald et al. conclude that for larger scenes, other approaches like incremental or parallel rebuilds may be needed.

- For small and medium-size animated scenes, the system reach interactive rates. For example, a scene with 80K triangles can be rendered at 16fps (frames per second) without shading, and at 9fps with shading.

As a conclusion, we can see that regular grids with coherent ray traversal is competitive with other data structures and traversal methods, like kd-trees.

7 Improved BSP and kd-trees

The Binary Space Partitioning (BSP) tree is a data structure to represent a scene using a binary tree [8, p. 66]. The space is cut by a *hyperplane*, a line in 2D and a plane in 3D, and the hyperplane is not necessarily orthogonal to the axis. Figure 14 shows the construction of a simple BSP. BSP has the advantage that they expedite the process of visibility determination [8, p. 67], but also have the problem that it shape is determined by the order of insertion, and that in the worst case, they can degenerate in a chain [8, p. 235]. There are heuristic to avoid the worst case and Samet [8, p. 235] says that for a 3D space, the best known method produces a tree with a expected height of $O(n \log^2 n)$, in $O(n \log^3 n)$ time.

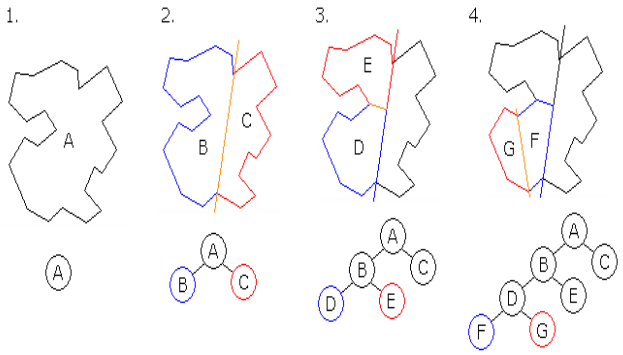
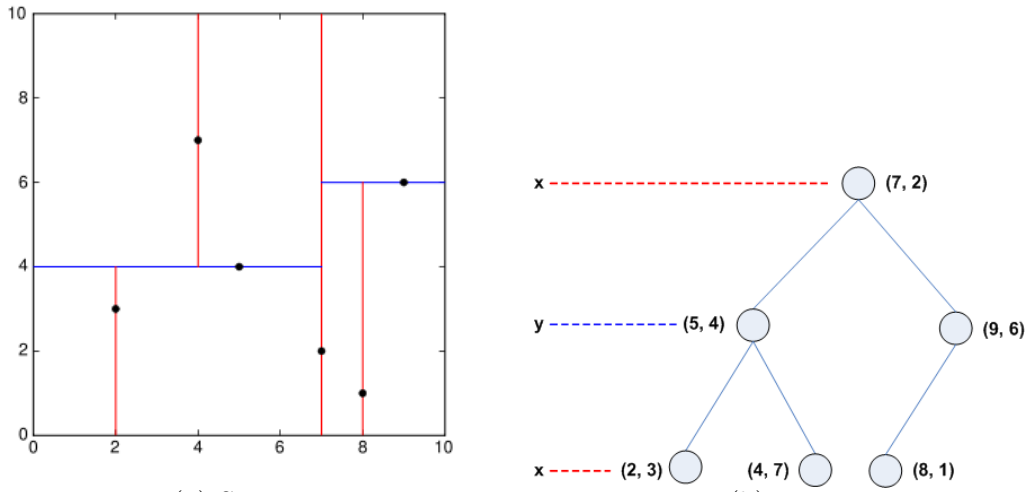


Figure 14: The construction of a simple BSP in 2D. On each step, the space is divided in two subspaces. The upper side has the space and the lower side the binary tree representation. Copied from Wikipedia, http://en.wikipedia.org/wiki/Image:Binary_space_partition.png, under the terms of the GNU Free Documentation License, Version 1.2.

The kd-tree is a generalization of the BSP. The kd-tree is also a binary tree, with the characteristic that each level represents a division in one coordinate axis. For example, for 3D scenes, the root stores the division in the X axis. Each children of the root (second level) stores the division of the two subspaces, created by the root, by the Y axis. The next level does the same for the Z axis, and so on. In Figure 15 there is an example for a kd-tree in 2D.

Samet's book [8] has an extensive review of kd-trees. Construction of a kd-tree takes $O(n \log n)$ time, insertion and search in $O(\log n)$ time. kd-trees are very popular, not only in Computer Graphics, because they are very efficient in *range search*, where questions like "retrieve all points that are within X distance from point P", are answered [8, p. 55].



(a) Space (b) Tree

Figure 15: A kd-tree in 2D. Copied from Wikipedia, http://en.wikipedia.org/wiki/Kd-tree, under the terms of the GNU Free Documentation License, Version 1.2.

7.1 Coherent ray tracing

Wald et al. [15] used an axis-aligned BSP in their implementation of an optimized ray tracer called RTRT. Taking advantage of coherence techniques and modern graphics hardware, RTRT outperformed other available ray tracers by more than an order of magnitude.

7.1.1 Implementation

Using Intel SSE (Streaming SIMD Extensions) extensions, the authors are capable of tracing four rays in parallel in a Pentium-III CPU. The traversal of the BSP structure is very simple. Each ray is compared in parallel with the voxel splitting plane. If the ray is at the left of the splitting plane, the left children is visited. If the ray is right of the splitting plane, the right children is visited. If the ray cuts the splitting plane, the ray is clipped and each children is visited for each segment.

RTRT has other optimizations. In fact, most of the design decisions were took with the Pentium III architecture in mind. For example, the data structure for storing triangles, the only supported object, occupies 48 bytes in order of taking advantage of the cache line alignment. The problem in these systems is that they are not very portable.

7.1.2 Results

Wald et al. compared RTRT with other two free available ray tracers, POV-Ray and Rayshade. RTRT run time was between 11 to 15 times faster than the other two ray tracers. Then, they compared RTRT with three hardware rasterization engines, SGI Octane, SGI Onyx and a dual processor PC running SGI Performer. RTRT showed better perfomance than the three rasterization engines, for scenes with 1 million triangles and more.

Finally, Wald et al. built a small distributed ray tracing system with five dual processor Pentium-III PCs, connected by a switched 100-Mbit Ethernet. With this arrangement, the authors obtained interactive frame rates between 2.4 and 7.7 frames per second.

7.2 Multilevel ray tracing

The Multi-Level Ray Tracing Algorithm (MLRTA), proposed by Reshetov et al. [17] is a new approach that combines the tracing of coherent rays with an improved kd-tree data structure. The improvements include new criteria to stop the subdivision of a voxel, and the definition of "entry points" in the kd-tree that avoid starting always the traversal from the root. The result is that MLRTA performance allows interactivity for complex scenes on ordinary desktops.

7.2.1 Results

The authors compared its own ray tracer with and without MLRTA. The conclusion is that MLRTA reduced between 1.3 and 3 times the number of traversal steps required.

8 Parallel ray tracing

Since ray tracing algorithm computes the color of each pixel independently of each other pixel, one could think it is easy to parallelize. But there are some issues, mainly because all the pixels depends on the same scene. In a multicomputer environment, where each computer has its own memory, the scene could be replied in each computer wasting space. Another option is divide the scene, and store it distributedly. This second option arises the problem of communication among the distinct processors, when a ray passes from one segment of the scene to another. A shared memory computer has not these problems. And for this reason it seems the most promising way to parallelize ray tracing [18, 19].

Another source of parallelism is the new technology in CPUs that incorporates superscalar technology, e.g. the Intel i960[5], and new GPUs (Graphics Processor Units), e.g. from NVidia[6], that allows to discard work load from the CPU into the GPU and running in parallel. As an example, in section 7.1, how Wald et al. [15] used the Intel SSE extensions to trace four rays in parallel.

Reinhard [19] says that three different types of data scheduling have been applied to ray tracing:

- Demand driven. If the scene can be loaded in every processor memory, or if the memory is shared, the master processor replicates all the data and divide the screen in disjoint areas. Each processor renders one or more regions, and the master assembles the final image. On the other hand, if there is no shared memory, and the scene is bigger than the local processor memories, then a efficient inter-processor communication must be need. Reinhard says that some of this communicating problems can be alleviated with a careful design that includes considerations for object, image, and ray coherence.

[5] http://www.intel.com/design/i960/superscalar.htm.
[6] http://www.nvidia.com/page/home.html.

- Data parallel is another option when the scene does not fit in local memory. In this approach there is no master processor, and the scene is divided equally between the processors. The shading is done by the processor which started a ray. If the ray needs to continue in a part of the scene belonging to another processor, the origin processor sends the ray data to the destine processor. Later, the destine processor will return the intermediate shading color to the origin processor, and this processor will write the final pixel color.

  One problem is that there is no a general method to get a reasonable load balance. If the objects are uniformly distributed, a division by area would be the best. But if there are clusters of objects, then the best method would be assign each processor roughly the same number of objects. And even with these considerations, there is no guarantee of a good load balance, because for example, a shiny metallic objects needs more work than an opaque object.

- Hybrid scheduling tries to combine the two other approaches. Reinhard opines that data parallel is best when each processor needs a big amount of data, or when is difficult to predict which data is required to complete a task. For tasks more computationally intensive and which do not require much data, the demand driven is a better approach

9 Conclusion

After this study, we can see that the basic algorithms in ray tracing are well established and known. It seems that the complexity, at least for now, is bounded by the factor $O(\log n)$ which is intrinsic to tree operations. New methods and approaches try to reduce the running time, not the complexity, sometimes by tweaking the data structures, and sometimes by exploting new hardware capabilities.

Another conclusion is that there are not a general method nor data structure that works for every case and every type of scene. As Klimaszewski and Sederberg [10] wrote, maybe the answer is in specialization.

References

[1] Alan Watt. *3D Computer Graphics*. Addison-Wesley, Harlow, England, third edition, 2000.

[2] Boris Aronov, Hervé Brönnimann, Allen Y. Chang, and Yi-Jen Chiang. Cost-driven octree construction schemes: an experimental study. In *Proceedings of the 19th annual symposium on Computational Geometry*, pages 227–236, San Diego, CA, June 2003. Association for Computer Machinery.

[3] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Morgan Kaufmann, San Francisco, CA, 1989.

[4] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, Harlow, England, 1992.

[5] Eric Haines. *Essential Ray Tracing Algorithms*, pages 33–77. In Andrew S. Glassner, editor. An Introduction to Ray Tracing. Morgan Kaufmann, San Francisco, CA, 1989.

[6] Eduard Gröller and Werner Purgathofer. Coherence in computer graphics. Technical Report TR-186-2-95-04, Technische Universität Wien, Wien, Austria, March 1995.

[7] P. González and F. Gisbert. Object and ray coherence in the optimization of the ray tracing algorithm. In *Proceedings of the Computer Graphics International (CGI) 1998*, pages 264–267, Hannover, Germany, June 1998. IEEE Computer Society.

[8] Hanan Samet. *Foundations on Multidimensional and Metric Data Structures.* Morgan Kaufmann, San Francisco, CA, 2006.

[9] Robert Endl and Manfred Sommer. Classification of ray-generators in uniform subdivisions and octrees for ray tracing. *Computer Graphics Forum*, 13(1):3–19, 1994.

[10] Krzysztof S. Klimaszewski and Thomas W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, pages 42–51, January-February 1997.

[11] Kyu-Young Whang, Ju-Won Song, Ji-Woong Chang, Ji-Yun Kim, Wan-Sup Cho, Chong-Mok Park, and Il-Yeol Song. Octree-R: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, December 1995.

[12] Boris Aronov, Hervé Brönnimann, Allen Y. Chang, and Yi-Jen Chiang. Cost prediction for ray shooting. In *Proceedings of the 18th annual symposium on Computational Geometry*, pages 293–302, Barcelona, Spain, 2002. Association for Computer Machinery.

[13] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Proc. Eurographics '87*, pages 1–10, Amsterdam, Holland, August 1987. Eurographics Association.

[14] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. In *Proceedings of the ACM SIGGRAPH 2006*, pages 485–493, Boston, MA, August 2006. Association for Computer Machinery.

[15] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–165, 2001.

[16] Erik Reinhard, Brian Smits, and Charles Hansen. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the 11th Eurographics Workshop on Rendering Techniques*, pages 299–306, Brno, Czechia, 2000. Eurographics Association.

[17] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *Proceedings of the ACM SIGGRAPH 2005*, pages 1176–1185, Los Angeles, CA, July-August 2005. Association for Computer Machinery.

[18] Steven Parker, William Martin, Peter-Pike Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *Proceedings of the 1999 symposium on Interactive 3D Graphics*, pages 119–126, Atlanta, GA, 1999. Association for Computer Machinery.

[19] Erik Reinhard. *Parallel Global Illumination Algorithms*, pages 89–132. In Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. Practical Parallel Rendering. A. K. Peters, Natick, MA, 2002.