A Ray Tracing Accelerator Based on a Hierarchy of 1D Sorted Lists

Alain Fournier
Pierre Poulin

Department of Computer Science
University of British Columbia
{fournier | poulin}@cs.ubc.ca

ABSTRACT

Since the introduction of ray tracing as a rendering technique, several approaches have been proposed to reduce the number of ray/object intersection tests. This paper presents yet another such approach based on a hierarchy of 1D sorted lists. A bounding box aligned with the axes encloses an object. The coordinates of each bounding box are ordered in three sorted lists (one for each axis) and are treated as *events*. Traversing a scene with a ray consists of traversing each sorted list in order, intersecting an object only when for this object a first event has been encountered (entered) in every dimension before a second event has been encountered (exited) in any dimension. To reduce the number of events (entries and exits) traversed, a hierarchy of sorted lists is constructed from a hierarchy of bounding boxes. The results are favourable for scenes ranging from moderate to high complexity. Further applications of the technique to hardware assist for ray tracing and to collision detection are discussed.

RESUME

Depuis l'introduction du lancer de rayon comme technique de synthèse d'image, plusieurs approches ont été proposées pour réduire le nombre de tests d'intersection entre les rayons et les objets. Cet article présente une approche basée sur une hiérarchie de listes ordonnées dans chaque dimension. Chaque object est entouré par une boîte alignée avec les axes. Les coordonnées de chaque boîte sont ordonnées dans trois listes (une pour chaque axe) et sont traversées comme des *évènements*. Traverser une scène avec un rayon consiste à traverser chaque liste en ordre, intersectant un object seulement s'il a été rencontré par un premier évènement (d'entrée) dans toutes les dimensions avant un deuxième évènement (de sortie) dans n'importe quelle dimension. Pour réduire le nombre d'évènements (entrées et sorties) traversés, une hiérarchie de listes est construite à partir d'une hiérarchie de boîtes. Les résultats sont favorables pour des scènes de complexité moyenne à élevée. D'autres applications de la technique pour l'accélération par matériel du lancer de rayon et pour la détection des collisions sont aussi présentées.

KEYWORDS: ray tracing, acceleration, culling, space subdivision, bounding volumes, collision detection.

INTRODUCTION

Whitted [whit80] introduced ray tracing as a rendering technique. In its naive implementation, each ray must be intersected with each primitive in a scene. This approach is feasible only for scenes of modest size, and much research has focused on ways to make this technique more efficient for complex scenes.

Arvo and Kirk [arvo89] give a good survey of ray tracing acceleration techniques. They classify acceleration techniques in three categories: (1) faster intersections, (2) fewer rays and (3) generalised rays. Faster intersections are obtained by (1.a) reducing the intersection cost between a primitive and a ray or by (1.b) reducing the number of ray/object intersection tests. Our ray tracing acceleration belongs to this later classification (1.b). This category includes various space subdivision schemes, directional techniques and hierarchies of bounding volumes.

We can divide 3D spatial subdivision algorithms into two classes: uniform and non-uniform. Uniform subdivision [fuji86] [aman87] has the advantage of being easy to implement and the cost of traversing each element of the regular grid is very small. Unfortunately, the performance degrades when there are too many voxels because many empty voxels might be traversed and because of the cost of storage of the voxels. The performance degrades also when there are too few voxels because of the possibility of having a large number of objects to intersect within a single voxel. Moreover, there is not yet any good criteria to determine the optimal or even near optimal grid subdivision for a given scene.

Non-uniform space subdivision can adapt its resolution to the complexity of a scene and therefore it is less sensitive to the problems of uniform space subdivision. Unfortunately, traversing a non-uniform structure is more expensive than traversing a regular grid. Various non-uniform subdivisions have been used, including irregular grids [giga88], octrees [glas84], BSP trees [kapl85] and k-d trees [fuss88]. Snyder and Barr [snyd87] propose a technique that can be used along with most of these techniques. They surround each ray by a box to check against the bounding volumes of the objects to intersect. Some researchers have also proposed to use combinations of uniform and non-uniform subdivisions to alleviate the disadvantages of each structure while trying to benefit from their respective advantages.

Directional techniques rely, as the name indicates, on the direction a ray takes. These directions are classified by *direction cubes* subdivided regularly or adaptively. The cubes can be located at specific point locations as for the light buffer [hain86], onto surfaces as for ray coherence [ohta87] or in volumes as for ray classification [arvo87].

Other algorithms have been using hierarchical bounding volumes to reduce the number of ray/object intersection tests. Rubin and Whitted [rubi80] were the first to use hierarchies of bounding volumes in ray tracing. Weghorst *et al.* [wegh84] studied criteria for choosing efficient bounding volumes for ray tracing. Kay and Kajiya [kay86] use slabs as tighter bounding volumes. Charney and Scherson [char90] use binary trees of bounding volumes to reduce the number of ray/bounding volume intersections. However each of these approaches (except [char90]) relies on sorting the intersections with sub-bounding volumes or primitives each time a bounding volume is entered. We propose to sort once as preprocessing the bounding volumes along three orthogonal axes. This, as we will see later, still requires some sorting for each level of the entered bounding volumes, but we claim that the savings in unnecessary intersections and sortings are valuable. The traversal also allows for a fast initialisation of secondary rays.

To make the description of our method clearer, we will start by describing the ray traversal without any hierarchy. Then, we will explain how we build automatically our hierarchies and how to make the required extensions to the ray traversal. Finally, we will compare our algorithm to uniform grid traversal to show its advantages and disadvantages, and discuss other possible developments.

TRAVERSING SORTED LISTS

Assume each object is surrounded by a 3D bounding box, a parallelepiped aligned with the axes. For each axis, this bounding box is delimited by two values in world coordinates that we call *events*. Three doubly linked *event lists* are constructed from the events of every bounding box in the scene. Each node in these lists contains the value of its event (world coordinate), an integer ranking this event, a pointer to its object and pointers to the previous and next events.

When a ray traverses a scene, it goes through a series of events determined by the event lists. Assume for now that the ray origin is outside of all three event lists. The first event in each X, Y and Z event lists are identified, according to the ray origin and direction. A Δt is associated with the distance traversed along the ray such that

$$\Delta t[i] = \frac{(\text{event}[i] - \text{ray.origin}[i])}{\text{ray.dir}[i]}, \qquad i \in X, Y, Z.$$

For each next event in the three event lists, the Δt's are computed and the smallest is chosen and treated. After this event has been treated, only the Δt for the next event in the corresponding event list needs to be recomputed.

The ray owns a list of *active* objects in what we call a *ray list*. For an object to become *active*, an event has to occur for it in X, Y and Z. Treating an event corresponds to do the following:

```
if the object is marked processed for the current ray
      return
test the event status for the current bounding box
if the event is IN in the current dimension
      if the object is marked IN in every other dimension
            the object is added to the ray list
            the object is tested for intersection
else /* the event is OUT in the current dimension */
      the object is marked processed for this ray
      if the object is marked IN in every other dimension
            the object is removed from the ray list.
```

The event lists are traversed until the end of one of the lists is reached or until the next event has a Δt larger than the Δt of the closest intersection computed so far.

When a secondary ray (reflection, refraction, shadow) is started after an intersection, the ray list of its parent ray is first copied for this secondary ray. The next events must then be found. For each dimension, the next event is the same as the parent ray if the secondary ray has the same direction. Otherwise the next event is simply the previous event. Since the origin of the secondary ray and of the primary ray are different, the three Δt's need to be recomputed too. Finally, once this initialisation is done, before treating any event, the ray list is checked for the objects that are IN in every dimension. These objects will be tested for intersection and the smallest positive Δt will be kept in case of intersection. After this initialisation, the secondary ray can traverse the scene as previously described.

In the general case the camera position, which is the origin of the primary rays, can be inside the event lists. Its ray list also needs to be initialised. For that purpose, once per image, as preprocessing, the event lists are traversed, one at a time, from any end of each event list until the camera position is reached. No intersection is computed during this traversal. When a primary ray is shot, the same initialisation than for a secondary ray occurs, but the ray list of the camera is used as the ray list of the parent ray.

We implemented our algorithm as described in this section. As expected, we observed a great reduction in the number of ray/object intersection tests over testing every primitive. In fact, our ray traversal guarantees the minimal number of ray/object intersection tests if this number is solely based on the information provided by the 3D bounding volumes. Table 1 shows the results for two standard

Ray Traversal Technique			No Grid	3 Sorted Lists No Hierarchy		
Scene	Number Objects	Minimum Intersections	Time per ray (msec)	Intersections (per ray)	Events (per ray)	Time per ray (msec)
Spheres I	11	0.34	0.25	1.30	28.2	0.58
Spheres II	92	0.38	1.29	1.41	241.7	3.67
Spheres III	821	0.42	10.69	1.50	2048.8	29.75
Tetra I	4	0.32	0.12	2.07	4.9	0.27
Tetra II	16	0.30	0.20	2.72	19.2	0.51
Tetra III	64	0.29	0.54	3.29	78.8	1.41
Tetra IV	256	0.29	1.90	3.61	322.3	4.92
Tetra V	1024	0.28	7.30	3.77	1315.8	18.98
Tetra VI	4096	0.27	28.80	3.84	5354.1	75.58
Teapot	29	1.00	1.67	2.70	119.9	2.63

Table 1: No Hierarchy

test scenes [hain87] (top left and bottom right of figure 2) and Newell's teapot. We can observe that increasing the number of objects in these scenes leads to a fairly stable ratio of intersections for our ray traversal technique while the naive approach must test for intersection every primitive. For instance in "Spheres III", uniform grid traversal must perform 1955 times more ray/object intersection testing than the minimum while the ratio for list traversal is only 3.57. Therefore, we met our goal of reducing the number of ray/object intersection tests. Unfortunately, treating an event represents a significant portion of intersecting a bounding volume. Moreover a large number of events is treated as illustrated by the *Events* column for each scene. These events account for the major portion of rendering time. So much in fact that as the number of objects increases, traversing their respective events completely overwhelms the benefits of the reduction of ray/object intersection tests.

To reduce the number of events to traverse, we adopted a strategy based on a hierarchy of event lists. The attraction of the hierarchical structure lies in its low sensitivity to the complexity of a scene. Assuming a well balanced tree to represent a hierarchy, adding an order of magnitude more objects to a scene leads only to a logarithmic increase of the rendering complexity. In the next two sections, we will explain how we build our hierarchies and what are their effect on the performance of our implementation.

BUILDING A HIERARCHY OF SORTED LISTS

Building an optimal hierarchy can lead to a *logarithmic* growth of the number of intersection calculation per ray as a function of the number of objects. As such, this approach is very appealing for any ray tracer that might need to render thousands, if not millions of primitives. Building an optimal hierarchy, however, can be very costly as the number of objects increases, and often we do not know enough about the statistics of the scene to define adequate criteria of optimality. Most criteria are based on the randomness of the direction of rays, but this is not met in most scenes where a large proportion of rays emanates from a single position (the camera position) or are directed towards a few positions (light sources). Nevertheless, some simple criteria are useful to roughly estimate the relative cost of different hierarchies.

Various criteria have been proposed to evaluate hierarchies. Weghorst *et al.* [wegh84] included the cost of intersection and the tightness of fit of the bounding volumes. Kay and Kajiya [kay86] relied mostly on surface area and closeness between objects. Goldsmith and Salmon [gold87] used surface area and conditional probability to intersect the enclosed objects.

Building a hierarchy of bounding volumes can be done manually, automatically or both. For instance, a single bounding volume containing an entire hierarchy can be added automatically while its content will remain in the same structure. Our automatic hierarchy of bounding volumes is built in a *top-down* fashion using proximity to current volumes. It is very similar to Glassner's [glas88] construction except for the fact that we permit overlapping between bounding volumes. This avoids splitting objects or duplicating them in two or more volumes. This also allows for easier merging of hierarchies.

First the bounding volume enclosing the entire scene is computed. If this bounding volume encloses more than a predefined number of objects, it is subdivided. To subdivide this volume, a regular grid is laid on it in order to partition it into smaller volumes. It is important to note here that the resolution of the grid can be different along different axes. The center of mass of the bounding volume of each object is computed and the object is included in the grid element countaining its center of mass. For each grid element, two kinds of boundaries are maintained: the boundaries defined by the regular grid, and the boundaries defined by the bounding volumes of the included objects.

At the end of one pass, the latter becomes the permanent boundaries of the bounding volume. It is therefore conceivable to end out with empty (in which case they are removed from the structure) or overlapping bounding volumes.

This subdivision scheme is applied recursively until the number of objects in a bounding volume is smaller than a specified threshold or all the remaining objects are added to a single bounding volume. In this case, at least one object must cover a large portion of the bounding volume. These objects are identified and put in a special grid element for this bounding volume. A bounding volume has then a predefined maximum number of children, but the actual number varies; as we have seen the empty volumes are removed from the data structure. Once the entire hierarchy is built, the event lists are built for every bounding volume. In the following, to simplify the terminology we will use the term *objects* to designate the bounding volumes of single objects (the leaves of our tree), and *bounding volumes* for the non-leaf nodes.

TRAVERSING A HIERARCHY OF SORTED LISTS

In order to use this hierarchy in our ray traversal scheme, we must modify a few elements of our algorithm. First, both objects and bounding volumes can appear in the ray list; they have to be treated differently. When an object is entered in its three dimensions, it is tested for intersection. When a bounding volume is entered in its three dimensions, it is *opened*. Opening a bounding volume corresponds to traversing the list of events of the bounding volume in every dimension but the current one. These events are traversed until the Δt of the next event is greater than the Δt in the current dimension. The next events in every dimension will then be added to the list of next events according to their Δt's.

Since a list of events is sorted in advance only for each direct children of its bounding volume, we need be able to chose efficiently the next events among the candidates from each currently opened bounding volume. The *list of next events* in one dimension is a list ordered by the Δt's of its events. When a bounding volume is opened, its next events are the OUT events in every dimension. The events within the bounding volume must also be treated, which explains the need for a list of next events per dimension, with one node for each opened bounding volume.

When a bounding volume has been entered in every dimension, the next OUT event for itself requires a clean-up of the list of next events. Every next events for the other dimensions that correspond to an object/bounding volume of the current closing bounding volume must be removed from the list of next events.

To better understand the hierarchical ray traversal, consider the 2D example of figure 1. The ray starts at origin O. The first event encountered (event 1) is IN in X for the bounding volume. This event is followed by event 2 in Y. Since we are in 2D and both dimensions are IN, we need

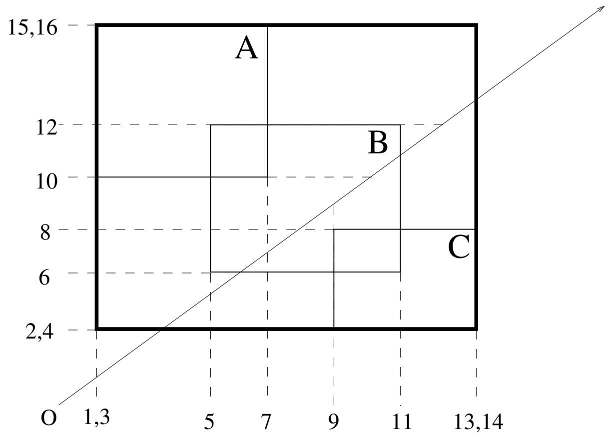

Figure 1: Ray hierarchy traversal

now to open this bounding volume and to add it to the ray list. It contains three objects A, B and C. Since we entered in Y, we need to update the events only in X. We traverse event 3 IN in X from object A and event 4 in Y from object C. Then 5 in X and 6 in Y requires an intersection test of object B. If an intersection is found that has a Δt smaller than the next event (7), then the ray traversal is stopped. If the Δt is larger than the one of event 7, since the intersection must be along the ray and within object B, we are guaranteed the Δt must be smaller than the Δt of event 11 and therefore, the ray traversal will stop before event 11 is treated. If no intersection is found, the traversal continues with event 7. Event 7 is an OUT in X for object A with no IN in Y. So object A is marked as processed for this ray. Similarly when event 8 is treated, object C is marked processed. If we look at the list of next events at this point, we find in X the next events being 9 (IN for object C) and 14 (OUT for bounding volume) in that order. In Y, we have 10 (IN for object A) and 16 (OUT for bounding volume). At events 9 and 10, objects C and A are not treated since already marked processed. After event 11, object B is OUT, so it is removed from the ray list and marked as processed. At event 12, B is not treated. Object C is not treated again at event 13 in X and at event 14, the bounding volume is exited. It is marked as processed and removed from the ray list. While closing this bounding volume, every event associated with objects A, B and C will be removed from the list of next events (i.e. 15 in Y for object A). The final event to be treated will be 16, an OUT in Y.

RESULTS AND COMPARISONS

We compared our list traversal algorithm to the uniform grid traversal described in [aman87]. Uniform grid traversal is a fairly standard acceleration technique we believe is stable enough from implementation to implementation to be considered a good frame of reference. Some comparisons between uniform grid traversal and other acceleration techniques can be found in [jeva89] [sung91] [subr91].

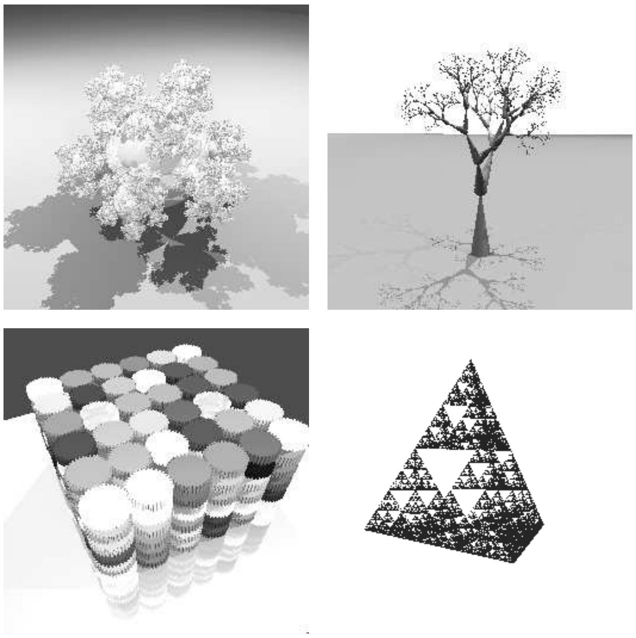

Figure 2: Haines' Testbed Scenes

Figures 3 through 10 give the timings and intersection computations for uniform grid traversal and list traversal for Haines' testbed scenes [hain87] illustrated in figure 2. Each scene is a sequence of increasing complexity. If we consider the number of intersections tested, except for the first 500 objects in the tetrahedron scene of figure 10, list traversal is always well below uniform grid traversal for a grid resolution of $30 \times 30 \times 30$. In fact in most cases, the curves for the number of intersection tests for list traversal follow quite nicely the curves for the minimum number of intersections. This is still true even if we include the number of bounding volumes opened. For two of these scenes (Spheres of figure 3 and Tree of figure 5), the rendering time of list traversal is significantly lower than uniform grid traversal. For these two test scenes, the list traversal curves approximate well a logarithmic rate of growth. For the other scenes however we do not have the same situation. In those scenes (Gears of figure 7 and Tetrahedron of figure 9), while the number of intersections tested remain much lower with list traversal, we observe that the number of events in list traversal is quite larger than the number of intersections in grid traversal. This large number of events treated is attributable to the characteristics of the hierarchy.

In Gears (figures 7 and 8), often several events occur at the same coordinate. For instance, each tooth (145 per gear) produces two events in Z at the same coordinates than the two faces of the gear. This means a ray has to traverse several layers of the hierarchy and treat several events in Z before finding an intersection. On the other hand, each occupied voxel has a maximum average of 22.8 polygons per voxel (for 31537 objects) and very few occupied voxels are traversed before an intersection is found. Since treating an event is a significant fraction of computing the intersection of a ray with a polygon, the time saved on intersection is more than offset by the cost of event processing.

In Tetrahedron (figures 9 and 10) , the maximum average number of objects per occupied voxel is 28.6 (for 65536 objects). Up to 14 intersection tests per ray are observed for grid traversal. Our automatic hierarchy ends out with a balanced tree with 8 levels, 4 nodes for each level with all objects in the lowest level. A ray coming from the eye would therefore have to traverse a minimum of 24 events before intersecting a triangle. Unfortunately, a ray traverses in average 3 times this number of events which makes grid traversal faster than list traversal.

It should be noted that especially in the last example, the cost of testing an intersection is quite low due to the simplicity of each primitive object (a triangle). In the case of more costly primitives (such as parametric patches), if treating an event is much faster than testing an intersection, there will always be a scene of a size such that list traversal will result in less computing time.

These results prove the efficiency of hierarchies of bounding volumes for some scenes but also its deficiency for others. The timings obtained in this paper are for a very simple algorithm to build automatically the hierarchy. Minor changes in the algorithm resulted in drastic differences in rendering time. This suggests, as many researchers previously working with hierarchies of bounding volumes concluded, the strong need for a better algorithm to build hierarchies and methods to evaluate their efficiency.

While the proposed testbed from Haines measures some important aspects of ray tracing, it is not typical of scenes used in computer animations. We compared list traversal to grid traversal with a scene taken out of the computer animation: *Around Again* [tess92]. This scene is made of 25426 triangles, 4610 quadrilaterals and 636 polygons with more than four vertices. Figure 11 shows the rendered scene. This scene is representative of a more typical scene with large meshes of small polygons (cans) positioned next to long polygons (walls and ground). On this scene, grid traversal tested as much as 618 times the minimum number of intersections. List traversal tested only 4.5 times this number and this resulted in 35% of the total rendering time of a $30 \times 30 \times 30$ grid traversal.

FURTHER DEVELOPMENTS

Traversing Multiple Directions

We described 3D traversal with three 1D sorted lists. It is also possible to use multiple sorted lists to do the traversal. In that sense, we could use an arbitrary number of directions with their sorted lists and therefore build convex slabs as in Kay and Kajiya [kay86]. The added cost is from adding new sorted lists to traverse, new dimensions

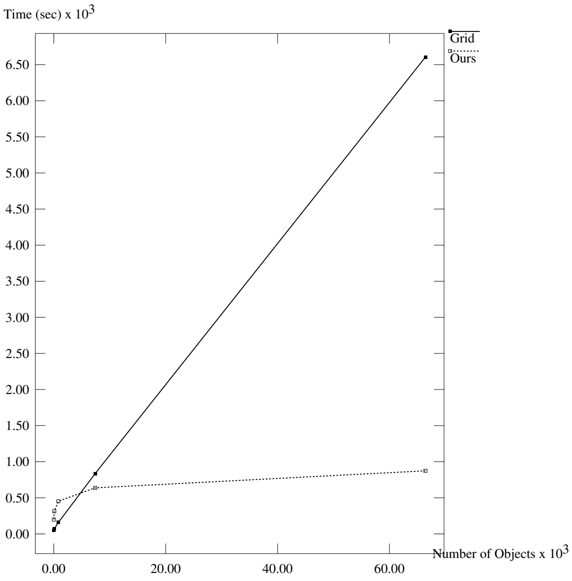

Figure 3: Sphereflakes' timings

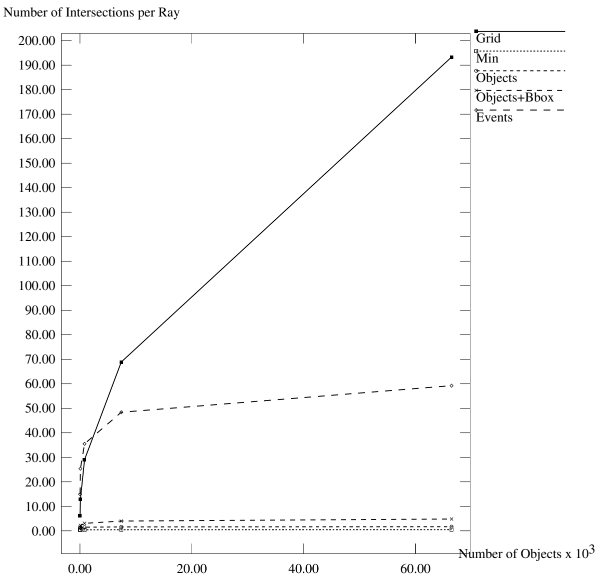

Figure 4: Sphereflakes' intersections

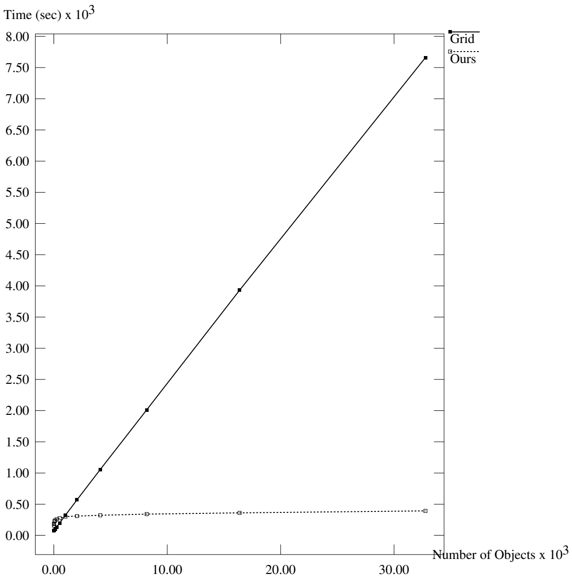

Figure 5: Tree's timings

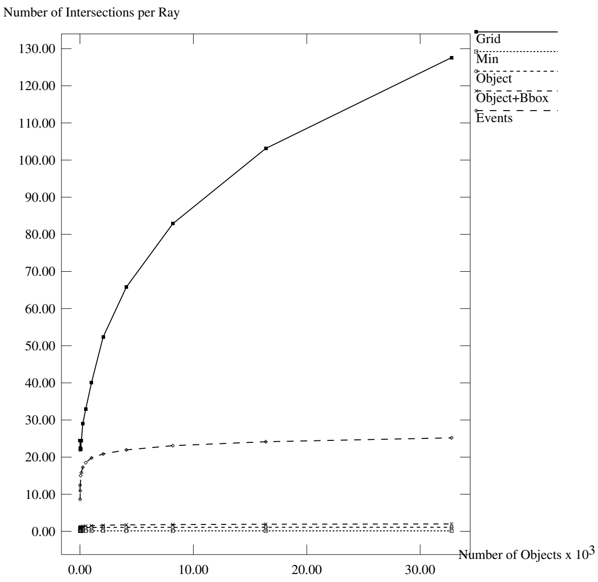

Figure 6: Tree's intersections

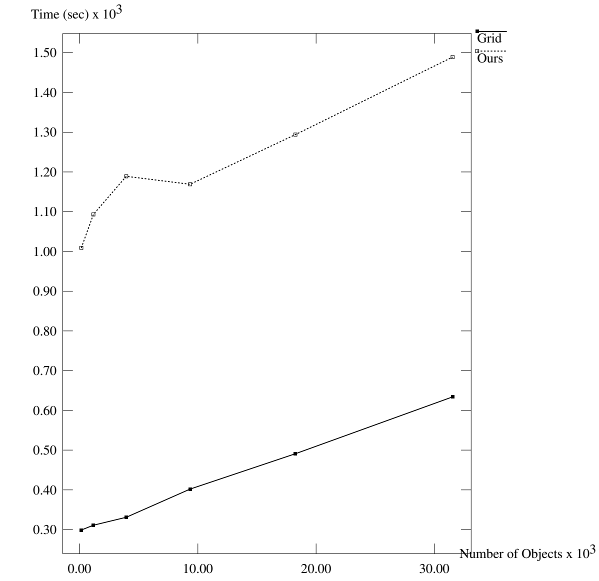

Figure 7: Gears' timings

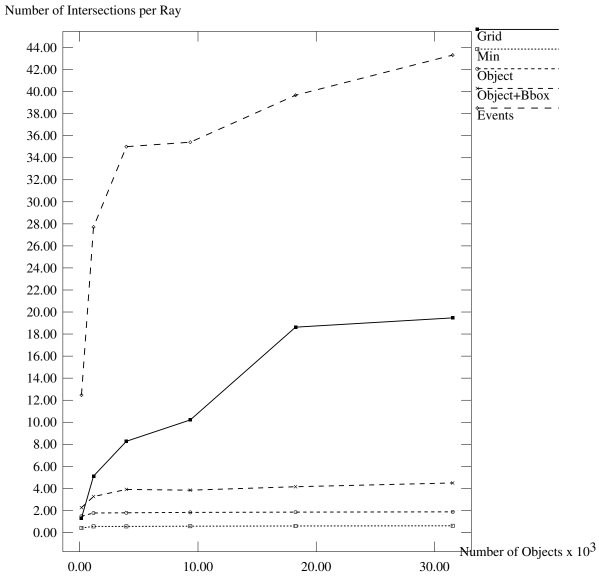

Figure 8: Gears' intersections

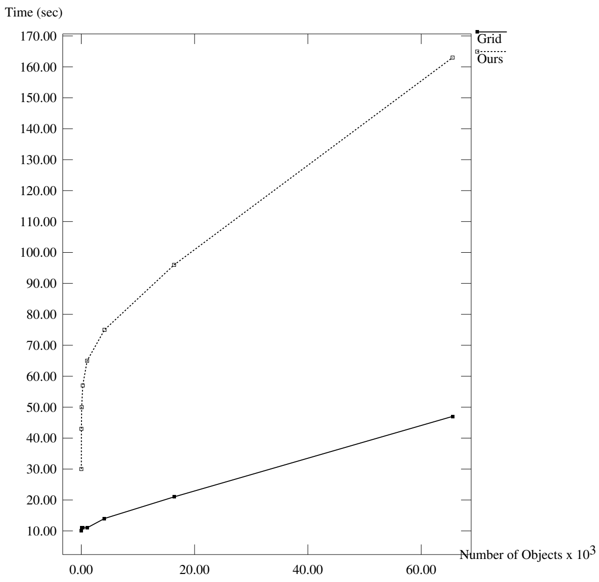

Figure 9: Tetrahedron's timings

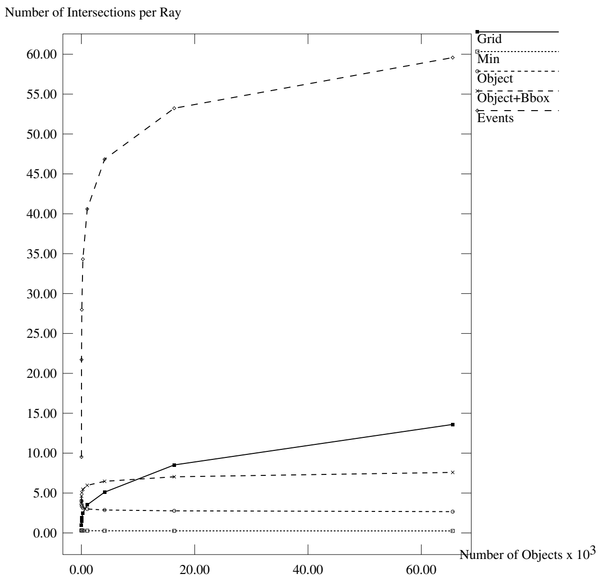

Figure 10: Tetrahedron's intersections

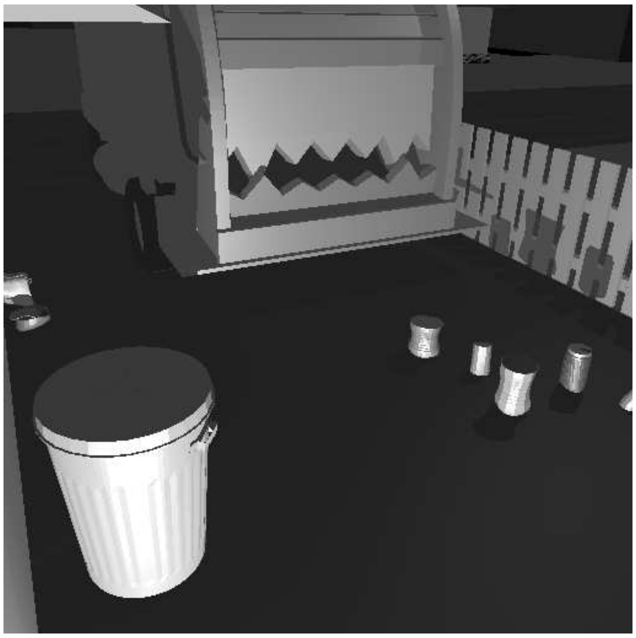

Figure 11: Scene out of the computer animation *Around Again*

to keep in the ray lists and new entries for the lists of next events. This cost would be offset only if the reduced area of the slabs versus regular 3D bounding boxes is significant, which has yet to be shown.

Hardware Ray Tracing

We used in our implementation sorted lists aligned with the axes in world coordinates. We can use different axes to speed up certain aspects of the ray traversal. If the traversal is done in the screen coordinate system, after the perspective transformation, all the primary rays have an origin at the center of pixels, and are parallel to the Z axis. For a ray, once the XY traversal is done to establish the ray list at the origin, only Z traversal takes place, and no Δt has to be computed. For the following pixel, the original ray list is inherited from the preceding one, with only a short X or Y traversal necessary to update it. After, the traversal is done only in Z. The whole process is easily done in firmware or hardware. Such an approach is compatible with the extensions included in the ZZ-buffer of Salesin and Stolfi [sale90].

It is also possible to transform the coordinates to speed up shadow determination for up to two directional light sources. To do so, two of the axes become the direction of each light source while the third is the direction away from the screen. Traversing the events to find the visible surface or the shadow determination is then done only for one list at a time.

Collision Detection

A ray list can be associated with the leading vertex of a bounding volume of an object. This leading vertex corresponds to the first vertex in the motion direction. To determine if a collision occurs while this object is moving in a 3D scene, we simply need to traverse the sorted lists in the motion direction. An event will be considered IN in one dimension if there is an overlap between the moving bounding box and the bounding box of the treated event. An intersection will be performed only when the event status is IN in every dimension. Once again, this scheme is suitable for hierarchies of bounding volumes.

CONCLUSION

We presented a speed up technique for ray tracing based on a hierarchy of 1D sorted lists. Unlike previously proposed bounding volumes approaches, our technique relies on pre-sorting the coordinates of the bounding volumes before rendering to avoid sorting intersections during rendering. This technique guarantees we always open a bounding volume or intersect an object in the order the ray traverses the scene. In that sense, if we rely only on the information provided by the bounding volumes of objects, we are guaranteed the minimum number of intersection tests with the objects, even for overlapping bounding volumes. The overlapping also allows for a more flexible handling of hierarchies.

The results demonstrate the interest of this technique for scenes of moderate to high complexity ($\geq$ 10,000 of objects). The graphs show how this technique is expected to be faster than uniform grid traversal as the number of objects grows. With the advent of faster CPUs and cheaper memory, we expect the number of objects used in common animations to reach easily this level of complexity. If ray tracing is considered for rendering these scenes, we believe hierarchies of some sort are the only realistic approach.

The next step in that direction should address the issue of creating more reliable hierarchies of bounding volumes. Many papers proposing hierarchies of bounding volumes to speed up ray tracing give various criteria to consider, but none seem to be implementing any of these criteria other than the proximity of the objects. In this paper, we described a technique similar to Glassner's [glas88] to create better hierarchies. However our experience shows how sensitive the rendering time can be to small changes in hierarchies. Better results in this direction are expected.

Finally, traversing a hierarchy of bounding volumes lends itself well to parallel processing. We are currently investigating the impacts of using our ray traversal method in this context.

ACKNOWLEDGEMENTS

We would like to thank Andrew Woo for his comments on early drafts of this paper, Chris Romanzin for his script conversion program and Markus Tessmann for his scene description taken from *Around Again*. We acknowledge financial support from NSERC, ASI, IBM Canada, UGF and the University of British Columbia.

REFERENCES

[aman87] John Amanatides and Andrew Woo. "A fast voxel traversal algorithm for ray tracing". *Eurographics '87*, pp. 3–10, August 1987.

[arvo87] James Arvo and David Kirk. "Fast Ray Tracing by Ray Classification". *Computer Graphics (SIGGRAPH '87 Proceedings)*, Vol. 21, No. 4, pp. 55–64, July 1987.

[arvo89] James Arvo and David Kirk. "A survey of ray tracing acceleration techniques". *An introduction to ray tracing*. pp. 201–262. Academic Press, 1989.

[char90] Mark J. Charney and Isaac D. Scherson. "Efficient Traversal of Well-Behaved Hierarchicial Trees of Extents for Ray-Tracing Complex Scenes". *The Visual Computer*, Vol. 6, No. 3, pp. 167–178, June 1990.

[fuji86] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. "ARTS: Accelerated Ray Tracing System". *IEEE Computer Graphics and Applications*, Vol. 6, No. 4, pp. 16–26, 1986.

[fuss88] Donald Fussell and K.R. Subramanian. "Fast Ray Tracing Using K-D Trees". Technical Report TR-88-07, U. of Texas, Austin, Dept. Of Computer Science, March 1988.

[giga88] Michael Gigante. "Accelerated Ray Tracing Using Non-Uniform Grids". *Proceedings of Ausgraph '90*, pp. 157–163, 1988.

[glas84] Andrew S. Glassner. "Space Subdivision For Fast Ray Tracing". *IEEE Computer Graphics and Applications*, Vol. 4, No. 10, pp. 15–22, October 1984.

[glas88] Andrew S. Glassner. "Spacetime ray tracing for animation". *IEEE Computer Graphics and Applications*, Vol. 8, No. 2, pp. 60–70, March 1988.

[gold87] Jeffrey Goldsmith and John Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing". *IEEE Computer Graphics and Applications*, Vol. 7, No. 5, pp. 14–20, May 1987.

[hain86] Eric A. Haines and Donald P. Greenberg. "The Light Buffer: A Ray Tracer Shadow Testing Accelerator". *IEEE Computer Graphics and Applications*, Vol. 6, No. 9, pp. 6–16, September 1986.

[hain87] Eric Haines. "A Proposal for Standard Graphics Environments". *IEEE Computer Graphics and Applications*, Vol. 7, No. 11, pp. 3–5, November 1987.

[jeva89] David Jevans and Brian Wyvill. "Adaptive voxel subdivision for ray tracing". *Proceedings of Graphics Interface '89*, pp. 164–172, June 1989.

[kapl85] M. Kaplan. "Space-Tracing: A Constant Time Ray-Tracer". *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*. pp. 149–158. July 1985.

[kay86] Timothy L. Kay and James T. Kajiya. "Ray Tracing Complex Scenes". *Computer Graphics (SIGGRAPH '86 Proceedings)*, Vol. 20, No. 4, pp. 269–278, August 1986.

[ohta87] Masataka Ohta and Mamoru Maekawa. "Ray Coherence Theorem and Constant Time Ray Tracing Algorithm". *Computer Graphics 1987 (Proceedings of CG International '87)*, pp. 303–314, 1987.

[rubi80] Steven M. Rubin and Turner Whitted. "A 3-Dimensional Representation for Fast Rendering of Complex Scenes". *Computer Graphics (SIGGRAPH '80 Proceedings)*, Vol. 14, No. 3, pp. 110–116, July 1980.

[sale90] David Salesin and Jorge Stolfi. "Rendering CSG Models with a ZZ-Buffer". *Computer Graphics (SIGGRAPH '90 Proceedings)*, Vol. 24, No. 4, pp. 67–76, August 1990.

[snyd87] John M. Snyder and Alan H. Barr. "Ray Tracing Complex Models Containing Surface Tessellations". *Computer Graphics (SIGGRAPH '87 Proceedings)*, Vol. 21, No. 4, pp. 119–128, July 1987.

[subr91] K. R. Subramanian and Donald S. Fussell. "Automatic Termination Criteria for Ray Tracing Hierarchies". *Proceedings of Graphics Interface '91*, pp. 93–100, June 1991.

[sung91] K. Sung. "A DDA Octree Traversal Algorithm for Ray Tracing". *Eurographics '91*, pp. 73–85, September 1991.

[tess92] Markus Tessmann. "Around again". SIGGRAPH '92 Electronic Theater, July 1992.

[wegh84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. "Improved Computational Methods for Ray Tracing". *ACM Transactions on Graphics*, Vol. 3, No. 1, pp. 52–69, January 1984.

[whit80] Turner Whitted. "An Improved Illumination Model for Shaded Display". *Communications of the ACM*, Vol. 23, No. 6, pp. 343–349, June 1980.