Ray Tracing Scenes of Varying Local Complexity

David A. J. Jevans
Department of Computer Science
University of Calgary
Calgary, Alberta
Canada T2N 1N4

January 6, 1990

Abstract

An adaptive algorithm for ray tracing scenes of varying local complexity is presented. Scenes are subdivided by an hierarchical 1D grid structure, and a fast traversal algorithm is used to trace rays through the scene. A cost function is used to determine the subdivision granularity at each level.

Results illustrating the relative performance of this algorithm, the octree approach, uniform space subdivision, and adaptive 3D grid subdivision are presented.

Key Words: ray tracing, space subdivision, adaptive, cost function

1 Introduction

Ray tracing [Whitted 80] [Kajiya 86] [Ward 88] is an elegant solution to realistic image synthesis which is becoming more widely used as a rendering technique as the demand for realistic images increases. It is also used in conjunction with radiosity methods for photo-realistic rendering [Wallace 87] [Wallace 89] [Sillion 89]. Ray tracing is a computationally expensive procedure often requiring hours of rendering time to produce a single image. The speeding up of ray tracing has been an important research issue since its inception. The two prime methods for speeding up ray tracing are a reduction in the number of rays traced [Heckbert 84] [Amanatides 84] [Shinya 87], and a reduction in the number of ray/object intersections performed. This paper presents a new spatial subdivision method for reducing the number of ray/object intersections.

1.1 Motivation

Ray tracing algorithms must perform well for the variety of scenes that they may encounter, and must also be able to efficiently render scenes containing areas of greatly varying local complexity. These types of scenes are the mainstay of computer animation, where there are often small and complex foreground objects placed in larger, less complex surroundings.

Adaptive algorithms are well suited to the rendering of these scenes, but the overhead incurred by traversing the data structures can often lead to poor performance. This paper details a spatial subdivision algorithm which can better adapt to local variations in scene complexity than previous 3D grid methods, yet does not incur the cost of deep tree traversal. A cost function which balances preprocessing and rendering time to determine the granularity of grid subdivision is described.

2 Previous Work

Rubin and Whitted [Rubin 80] determined that the majority of time in ray tracing was spent performing ray/object intersections. They proposed the use of bounding rectangular parallelepipeds around objects because rays can be more quickly intersected with bounding volumes than with the objects that they contain. If a ray does not intersect the bounding volume then the more expensive ray/object intersection tests need not be performed. Kay and Kajia [Kay 86] refined the idea of bounding object hierarchies to include arbitrarily tight fitting bounding volumes.

Spatial subdivision is alternative technique for reducing the number of ray/object intersections. Objects are sorted into areas of subdivided space through which rays are traced. Intersection calculations are performed for objects inside only those areas through which a ray passes.

The octree [Glassner 84] was presented as an adaptive method for subdividing scenes of varying local complexity. Complex scenes often require octrees of significant depth, and costly vertical tree traversal limits the speedup attainable.

Uniform subdivision was presented as an attempt to speed up the traversal process [Vatti 85] [Fujimoto 86] [Arnaldi 87] [Amanatides 87] [Scherson 87] [Cleary 88]. A scene is subdivided into a uniform 3D grid. Rays are traversed through the grid with an algorithm similar to a line drawing routine. This leads to very fast traversal, but the algorithm performs poorly with scenes of varying object density, as these scenes cannot be subdivided adequately due to memory restrictions and the increasing cost of traversing rays through empty voxels.

The adaptive 3D grid algorithm combines the octree and uniform subdivision approaches and outperforms them in many cases [Jevans 89]. Tree depth is minimal, usually less than 4, and horizontal traversal is fast due to the uniform grids. It is difficult to choose the correct subdivision granularity at each node, however, and limiting the depth of the tree is often the key to obtaining good performance.

3 Overview

Subdividing along a single axis at a time has several advantages over the 3D grid approach. Firstly, a 3D subdivision is not imposed on the entire scene or subscene. Secondly, it is easier to develop a cost function to determine the granularity of subdivision and to limit the depth of the tree. Thirdly, there can be a memory and traversal cost savings in scenes which are large and sparse, but which contain small areas of densely clustered objects.

The algorithm presented in this paper utilizes a 1D cost function for determining the subdivision granularity at each level. The traversal algorithm is similar to the uniform grid traversal algorithm, and vertical traversal costs have been minimized.

4 Subdivision

The scene's bounding box serves as the root voxel for the subdivision process. At each level of the subdivision, the current voxel may be subdivided along the x, y, or z axis. The implementation detailed in this paper alternates between the three axes according to tree depth, in much the same way as Kaplan's bintree algorithm [Kaplan 85]. The granularity of subdivision at each voxel is determined by a cost function (see section 6). An example of spatial subdivision is presented in figure 1.

4.1 Data Structures

When a voxel is subdivided, a 1D array of voxel pointers of size **granularity** is created, and all objects in the voxel are sorted into this array. The voxel also stores pointers to the lowest voxels above it in each of the x, y, and z directions.

Each subdivided voxel contains a pointer to a *traversal state*, which maintains the state of the variables used for horizontal voxel traversal. Voxels which are subdivided along the same axis at the same *absolute granularity*, determined by multiplying the granularities of the current and all parent voxels which are subdivided along this axis, share the same traversal state. See figure 2.

An array of traversal state pointers for all possible absolute granularities is maintained for each of the x, y, and z axes. When a voxel is subdivided, the traversal state table for its axis is consulted. If a traversal state for its absolute granularity has already been created, the voxel receives a pointer to it. If not, a new traversal state is created, and both the voxel and the traversal state table receive pointers to it.

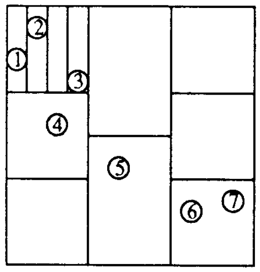
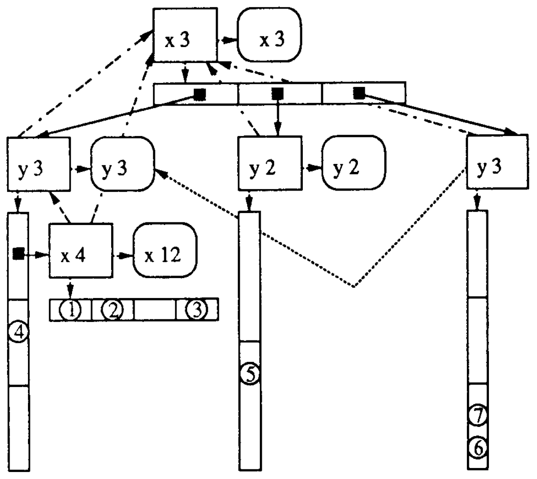


Figure 1: Subdivision



Legend

A voxel. Contains pointers to a traversal state, a voxel grid, and the previous voxels in the other axes.	x 4	The axis and granularity of the subdivision is indicated in the icon.
Traversal state. Traversal states of the same absolute granularity and axis are shared.	x 12	The axis and absolute granularity of the subdivision is indicated in the icon.
1D subvoxel grid. May contain subdivided subvoxels or objects.	▭▭▭	
Pointer to a subdivided subvoxel.	■——►	
Pointer to a voxel's members.	···········►	
Pointer to the lowest previous voxels in each axis.	— · — ·►	This has been left off the remainder of the diagrams in the interests of clarity.
An object.	○	

Figure 2: Subdivision data structure

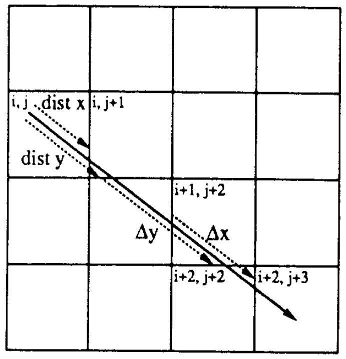


Figure 3: Cleary algorithm

The traversal state tables must be $\boldsymbol{max\_granularity}^{max\_tree\_depth}$ in size to accommodate all possible absolute granularities. If an implementation does not specify a limit on **max_granularity** or **max_tree_depth**, a hashtable could be used as a traversal state table instead of a large array.

Sharing traversal states typically reduces the number of states created by as much as 99%, and, as detailed in section 5.2.2, is used to significantly speed up the traversal process.

In order to ensure that only areas of a scene which are visible are subdivided, voxels are subdivided as they are first traversed by a ray, rather than in a preprocessing stage.

5 Traversal

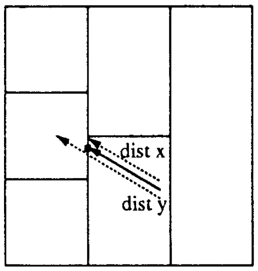
The traversal of a ray through a scene entails horizontal and vertical traversal of the subdivision data structure.

5.1 Uniform Grid Horizontal Traversal

Horizontal traversal of the data structure is similar to the uniform grid method developed by Cleary. A brief description of the Cleary algorithm follows.

The distance of a ray to the first interception of a voxel in each of the x, y, and z axes is stored in **dist[3]**. The distance along a ray between voxel intercepts in each direction is stored in Δ[3]. The three grid indices of the ray's starting voxel are stored in **index[3]**. A 2D example is illustrated in figure 3.

At each iteration of the algorithm, the distances to the next voxel intercept are compared. The index to the smallest of these, stored in **small**, indicates the direction of the ray's travel. Δ[**small**] is added to **dist[small]**, and **index[small]** is incremented or decremented by 1, depending on the sign of the ray's direction. Intersection tests are performed against the ray and any objects that lie inside the new voxel.



small is set to **x.**
The new voxel is subdivided along the **y** axis.
int_pnt[y] = ray_ori[y] + dist[small] * ray_dir[y]

Figure 4: Intersection of ray and voxel

Traversal of the 1D grids takes place as in the 3D grid algorithm, but the **dist**, Δ, and **index** variables for each dimension are accessed through the traversal states of the current x, y, and z subdivided voxels.

5.2 Vertical Traversal

Before horizontal traversal can take place, the data structure must be traversed vertically to its lowest subdivided level. If the current voxel is subdivided, the subvoxel in which the ray originates is determined. If this subvoxel is itself subdivided, the current voxel is set to point to it, and the algorithm repeats.

5.2.1 Determining the Initial Subvoxel

When a ray enters a subdivided voxel, the subvoxel in which it originates must be found before traversal can begin. The originating subvoxel's index is found by subtracting the voxel's minimum extents from the intersection point of the ray and the voxel, and dividing by the size of a subvoxel.

If the ray's origin lies inside the voxel, this becomes its intersection point. If not, an intersection point with the ray and the voxel must be found. The side of the voxel which the ray first intersects is known from the **small** term of the horizontal traversal. The intersection point with this side is calculated from the distance that the ray has traveled (figure 4).

5.2.2 Initialization of Traversal States

Whenever a ray enters a voxel, its traversal state must be initialized. Rays are identified by a unique **ray_id**, and the **ray_id** of the last ray through a voxel is stored in its state. If the **ray_id** of a ray matches the **ray_id** of a voxel's traversal state, then the ray has already been through the state, and only the **index** and **dist** variables need to be recalculated.

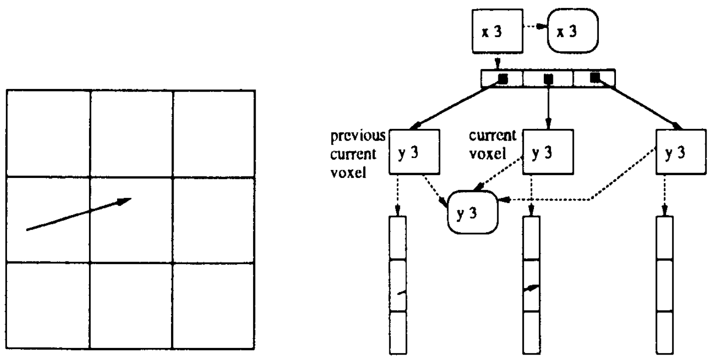


Figure 5: special case: uniform subdivision

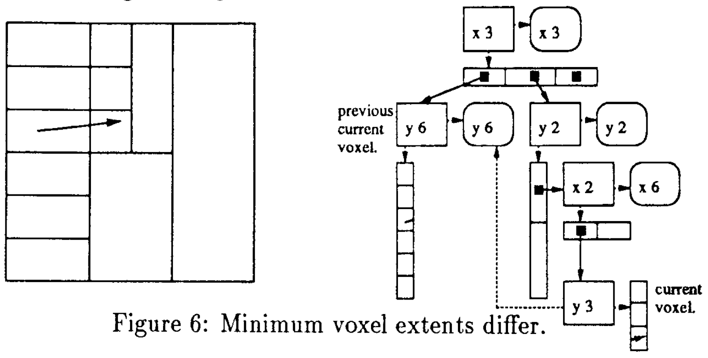


Figure 6: Minimum voxel extents differ.

A further optimization can be had by keeping a pointer to the previous current voxel in which horizontal traversal took place. This previous current voxel will be at the deepest part of the tree where horizontal traversal last took place. If the previous current voxel is subdivided in the same **axis** as the current voxel, then the *comparison voxel* points to the previous current voxel. If not, the the lower-most voxel above the previous current voxel which is subdivided in the **axis** direction, becomes the comparison voxel.

If the comparison voxel's state is the same as the current voxel's state, and the minimum voxel extents in the **axis** direction are the same for the two voxels, then the special case of a uniform subdivision has occurred, and *no* initialization needs to be done (figure 5).

If the current and comparison voxel's states are the same, but their minimum voxel extents differ, then the **index** of the ray must be computed (figure 6). This can be further optimized by noting that if the minimum extent of one voxel is equal to the maximum extent of the other (figure 7), the ray's **index** can be quickly calculated:

```
if (comparison_voxel→min_extent == current_voxel→max_extent)
    index = current_voxel→state→granularity - 1;
else if (comparison_voxel→max_extent == current_voxel→min_extent)
    index = 0;
```

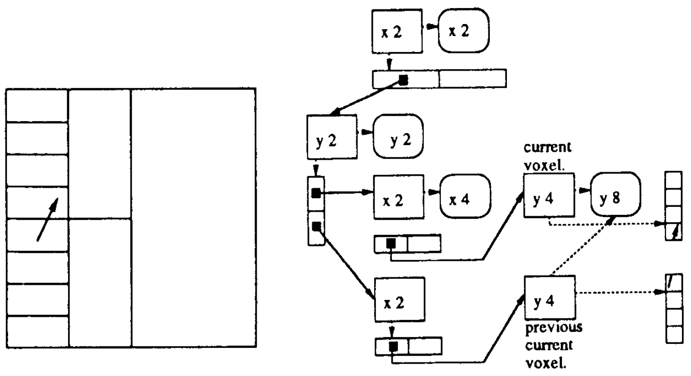


Figure 7: Minimum extent = maximum extent.

5.3 1D Horizontal Traversal

Whenever a new horizontal traversal is about to begin, the current traversal states, one for each of the x, y, and z, axes, must be found. The current states are those of the current voxel and the two voxels above it in the two other axes.

When a ray moves to a new subvoxel, one of three things may occur:

- The movement occurred at the lowest level of the tree (figure 8). If there are objects in the subvoxel, test for intersection. If not, and the subvoxel is subdivided, it becomes the current voxel.
- The movement occurred at a higher level of the tree (figure 9). Set the current voxel to be the one in which the movement occurred. If there are objects in the subvoxel, test for intersection. If not, and the subvoxel is subdivided, it becomes the current voxel.
- The ray exited a voxel grid (figure 10). Set the current voxel to the previous voxel in the direction of travel. If this is **NULL**, then the ray has left the scene entirely. If not, move the ray along one in the new voxel.

6 Cost Function

Arvo and Kirk [Arvo 89] note that "self-tuning" algorithms are necessary to efficiently render scenes with areas of varying local complexity. Previous methods have utilized non-intuitive user-defined parameters or incomplete sets of heuristics to control adaptive subdivision [MacDonald 89]. This section presents a cost function which is used to more intelligently control the subdivision.

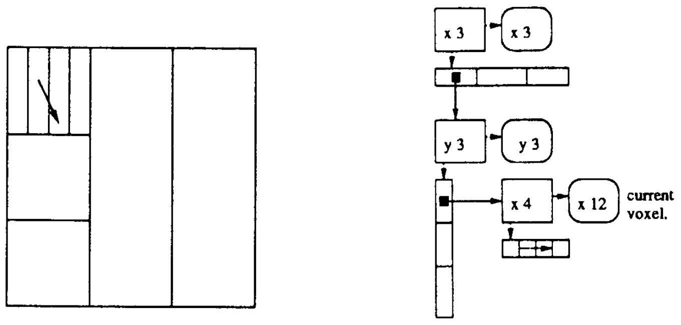


Figure 8: movement at lowest level of tree

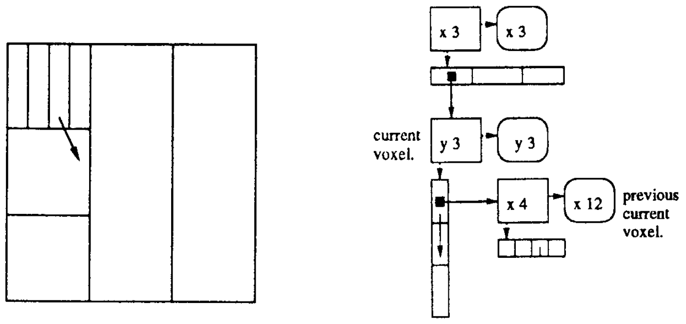


Figure 9: movement at higher level of tree

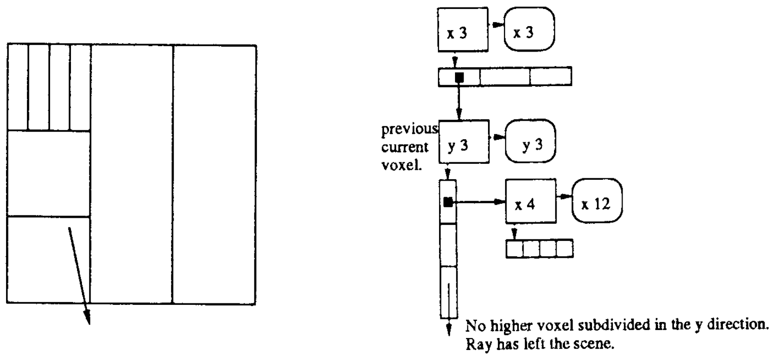


Figure 10: ray exited a voxel grid

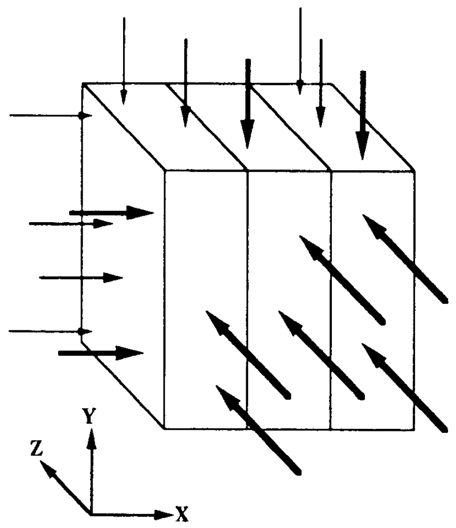


Figure 11: Rays through a subdivided voxel

6.1 Assumptions

Entire scenes are rarely uniformly subdivided, although small areas of scenes often are. Examples of this are tessellated surfaces, etc. We wish to develop a cost function which will quickly subdivide a scene into areas of uniform object distribution so that the uniform grid nature of the horizontal traversal algorithm can be made best use of.

In order to simplify the cost function, several assumptions are made:

- Assume uniform flux of rays through a voxel. The number of rays entering a voxel from a given side will be proportional to the surface area of the voxel's side [Stone 75] [Goldsmith 87].
- Assume that rays travel perpendicular to their plane of entry.
- Assume uniform distribution of objects.
- Assume rays pass through the entire voxel.

Since subdivision is along a single axis, and it is assumed that rays travel perpendicular to their plane of entry, only rays traveling perpendicular to the axis of subdivision will benefit from reduced ray/object intersections. Rays traveling along the axis of subdivision will pass through all the subdivisions, and will be intersected with all the objects.

In figure 11 the **x** axis has been subdivided. This example will be used in the following sections.

6.2 Code Timings

In order to develop a useful cost function, estimates of execution time for several operations must be found:

Intcost The cost of performing a ray/object intersection calculation.

Repcost The cost of repeating a ray/object intersection calculation.

Travcost The cost of iterating a ray through a subvoxel.

Vertcost The cost of initializing a ray to traverse through a subdivided voxel.

Subcost The cost of sorting an object into a subdivided voxel.

6.3 Cost Function for No Subdivision

Before the cost function can be evaluated, the number and mean size of the objects in the candidate voxel are determined. The size of an object must be truncated to the size of the voxel if the object is larger than the voxel, or the mean size of objects can be unfairly skewed.

Since the number of objects intersected by rays entering from the **yz** plane is constant, it is eliminated from the cost function.

The cost of traversing a ray through an unsubdivided voxel is proportional to the area of the **xy** and **xz** sides and the number of objects in the voxel:

int_cost = Intcost * (area_xy + area_xz) * num_objects

6.4 Cost Function for Subdivision

The cost of traversing a ray through a subdivided voxel is proportional to the area of the **xy** and **xz** sides and the number of objects per unit area:

int_cost = Intcost * (area_xy + area_xz) * num_per_unit

where:

num_per_unit = (num_objects * num_voxels_into) / num_subvoxels
num_voxels_into = (int)ceil(mean_object_size / subvoxel_size)

As the granularity of the subdivision increases, the objects become more tightly bound by the subvoxels, and the number of objects per unit area decreases to the minimum:

minimum_num_per_unit = num_objects * mean_object_size

As subdivision granularity increases, there is an increasing cost to rays traveling along the subdivision axis. This cost is made up of the time to iterate rays through the subdivisions, and the cost of repeatedly intersecting with objects that fall into more than one subvoxel. This cost is small due to the use of ray signatures (**ray_id**) which prevent performing a full intersection calculation of a ray with an object more than once [Arnaldi 87].

trav_cost = Travcost * area_yz * (num_subvoxels-1)
rep_cost = Repcost * area_yz * (num_voxels_into-1) * num_objects

All rays which pass through a subdivided voxel are subject to the cost of initialization:

ver_cost = Vertcost * (area_xy + area_xz + area_yz)

The total cost of traversing a ray through a voxel is:

total_cost = int_cost + trav_cost + rep_cost + ver_cost

6.5 Using the Cost Function

When a voxel is a candidate for subdivision, the cost function is calculated for the range of granularities **0** through **maximum granularity**. The granularity which results in the least cost is the one chosen for the subdivision level. Note that a maximum granularity is specified in order to limit the amount of memory required, and to encourage recursive subdivision at higher levels of the tree, where scene distribution is unlikely to be uniform. For this implementation a **maximum granularity** of 20 was used.

This cost function can suffer from the problem of over subdivision. If the number of rays passing through a voxel is small, and the time to do the subdivision is large, it is often better to leave the voxel unsubdivided. A breadth first traversal of a small sampling of rays before the full image is rendered provides a good estimate of the number of rays that will travel through a voxel.

The cost function is updated to be:

new_cost = Subcost * num_objects + estimated_num_rays * total_cost

Due to a generally good match between granularity and object size, the cost of sorting an object into a voxel grid is virtually independent of its size. If, however, the deviation of object sizes from the mean becomes very large, it may be necessary to take the size of objects into account when calculating the subdivision cost.

If a voxel is encountered during the ray tracing that was not traversed by the first sampling of rays, the number of rays through the voxel is estimated. The number of rays is estimated to be proportional to the surface area of the subvoxel relative to the surface area of its parent.

7 Results

A number scenes of varying complexity were rendered with similar implementations of several different space subdivision techniques.

1. Uniform subdivision. Grid granularity per side is set to $\sqrt[3]{num\_objects}$.
2. Octree.
3. Adaptive 3D grid subdivision.
4. Adaptive 1D grid subdivision.

The baseline for comparison is the *zero overhead* method. A scene is ray traced, and a file containing the identifier of the object intersected by each ray is generated. If a ray exits the scene, a **NULL** entry is recorded. The scene is then re-rendered using this information. A single intersection test, and any shading, is performed for each ray with a non-**NULL** entry. The time taken to do this rendering is the minimum possible, barring the use of ray coherence methods.

This implementation is written in C++, compiled with Gnu g++, and run on a SUN4/280 with 32 megabytes of real memory. All images were rendered at 512 by 512 resolution, with one ray per pixel. The graph in figure 12 illustrates the relative performance of these algorithms.

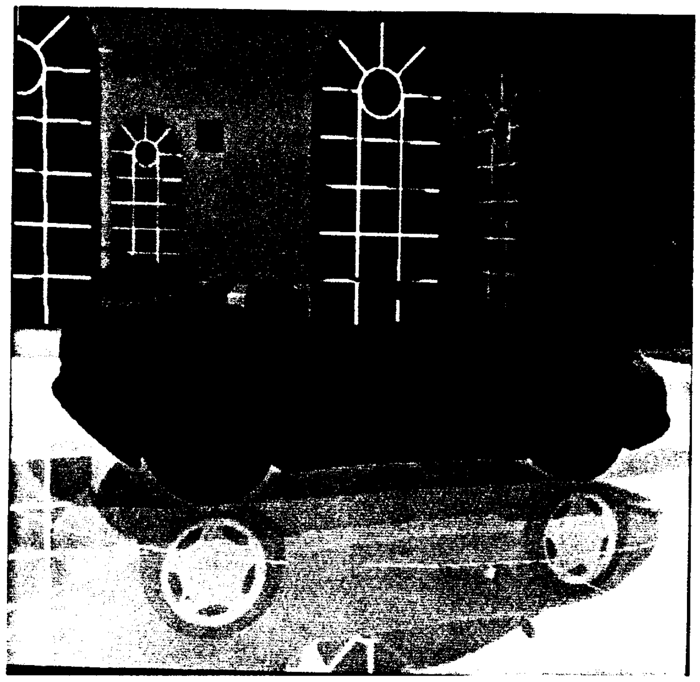
Note to referees: a performance increase of 30% to 50% is expected with the new release of the AT&T C++2.0 compiler, which is presently being installed at our site. Should this paper be accepted, the renderers will be re-compiled and the timings re-computed. The relative performance of the algorithms is expected to remain the same.

7.1 Discussion

pyramid This database is similar to that seen in previous papers. The performance of the four algorithms is similar due to the small number of polygons and the visibility of the entire scene.

lumpy A scene from a recent University of Calgary short film. The extents of the scene are large, but the majority of the polygons are clustered in a small area. The uniform subdivision method performs poorly due inadequate subdivision, and the octree method suffers due to the depth of the tree. The 1D method outperforms the adaptive 3D grid method owing to non-uniformity in the size of the x, y, and z extents of the areas of complexity.

drum Uniform subdivision yields poor performance due to the clustering of polygons. The octree method performs acceptably since the scene's extents are not large, resulting in a tree that is not overly deep. A drawback, however, is that memory usage is much higher than for the 3D and 1D grid methods. The 1D method outperforms the adaptive 3D grid method because it can better adapt to varying complexity, resulting in a subdivision tree of smaller depth.



car. 198,001 polygons. 1,762,000 rays.
304 minutes to render with 1D method.
Model courtesy of Alias Research.

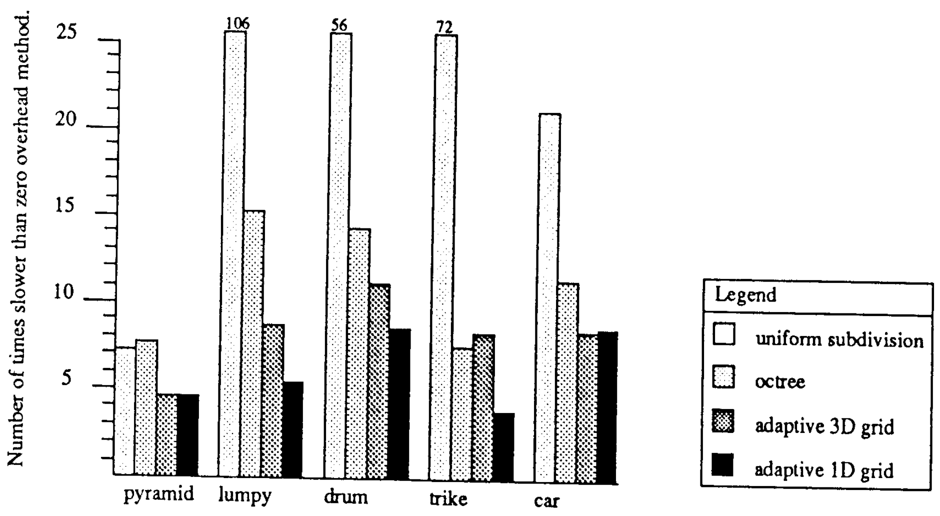


Figure 12: Results.

trike Uniform subdivision performs poorly because of the polygon clustering. The octree method performs well because the scene extents are not large. 1D subdivision outperforms adaptive 3D subdivision because the extents of the areas of complexity are irregular.

car Uniform subdivision performs poorly due to the complexity of the central object. The octree method performs reasonably well because the scene extents are not overly large compared to the central object. The adaptive 3D and 1D grid methods' performance are comparable because the scene extents are fairly regular, as are the extents of the complex object.

From these results, some characterizations of the four spatial subdivision methods compared in this paper can be made. Uniform subdivision performs poorly for scenes containing areas of local complexity, due to its inability to adequately subdivide space in these areas. Hybrid methods, such as that proposed by Snyder and Barr [Snyder 87], are required.

Octree subdivision performs well if the areas of local complexity are similar in size to the extents of the scene. If the extents of the scene are large and the complex areas are small, the octrees become deep, and vertical traversal limits the performance.

The adaptive 3D grid method performs well for a variety of scenes. It is fairly insensitive to the relative size of the scene extents and areas of complexity, but user intervention is sometimes required to set the **max_tree_depth** for a given scene. The 1D method has similar performance characteristics, but can better adapt to complex areas of irregular shape. It maintains a more constant level of performance for a variety of scenes with areas of local complexity. The cost function takes the number and size of objects into consideration when determining grid granularity, and effectively limits the depth of the subdivision tree.

8 Future Work

The introduction of this new algorithm and data structure opens many avenues of further research.

- Methods for determining axes of subdivision at each level.
- Cost functions which examine estimates of object distribution.
- Cost functions which examine combinations of recursive subdivisions.
- Balancing memory and rendering time for machines with restricted amounts of memory.
- Hybridizing with other methods.

- Applying shared traversal states to other spatial subdivision methods such as the octree.

9 Conclusion

A new algorithm for reducing the number of ray/object intersections has been introduced. The algorithm is adaptive in nature, yet traversal is fast. A cost function is used to intelligently control the subdivision. The algorithm is shown to perform well on a variety of scenes.

10 Acknowledgements

Thanks to Alias Research for providing many of the test scenes used in this paper, and to Brian Gaines and Andrew Pearce for their insights into this research.

References

[Amanatides 84] John Amanatides. Ray tracing with cones. In *Computer Graphics*, volume 18. ACM SIGGRAPH, 1984.

[Amanatides 87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. *Proc. Eurographics '87*, 1987.

[Arnaldi 87] B. Arnaldi, T. Priol, and K. Bouatouch. A new space subdivision method for ray tracing csg modelled scenes. *The Visual Computer*, 3(2):98–108, 1987.

[Arvo 89] J. Arvo and D. Kirk. A survey of ray tracing techniques. *An Introduction to Ray Tracing*, pages 201-259, 1989. Edited by Andrew S. Glassner.

[Cleary 88] John Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Visual Computer*, pages 65–83, July 1988.

[Fujimoto 86] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 1986.

[Glassner 84] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, pages 15–22, October 1984.

[Goldsmith 87] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE CG&A*, May 1987.

[Heckbert 84] P. Heckbert and P. Hanrahan. Beam tracing polygonal objects. In *Computer Graphics*, volume 18. ACM SIGGRAPH, 1984.

[Jevans 89] David A. J. Jevans. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89*, pages 164–172, 1989.

[Kajiya 86] James T. Kajiya. The rendering equation. In *Computer Graphics*, volume 20, pages 143–150. ACM SIGGRAPH, 1986.

[Kaplan 85] Michael R. Kaplan. The uses of spatial coherence in ray tracing. *SIGGRAPH '85 Course Notes 11*, 1985.

[Kay 86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *Computer Graphics*, volume 20, pages 269–278. ACM SIGGRAPH, 1986.

[MacDonald 89] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. In *Proceedings of Graphics Interface '89*, pages 152–163, 1989.

[Rubin 80] Steve M. Rubin and Turner Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[Scherson 87] L. Scherson and E. Caspary. Data structures and the time complexity of ray tracing. *The Visual Computer*, 3(4):201–213, 1987.

[Shinya 87] M. Shinya, T. Takahashi, and N. Seiichiro. Principles and applications of pencil tracing. In *Computer Graphics*, volume 21. ACM SIGGRAPH, 1987.

[Sillion 89] François Sillion and Claude Puech. A general two-pass method integrating specular and diffuse reflection. In *Computer Graphics*, volume 23. ACM SIGGRAPH, 1989.

[Snyder 87] John M. Snyder and Alan H. Barr. Ray tracing complex models with surface tessellations. In *Computer Graphics*, volume 21. ACM SIGGRAPH, 1987.

[Stone 75] L. Stone. Theory of optimal search. *Academic Press, New York*, pages 27–28, 1975.

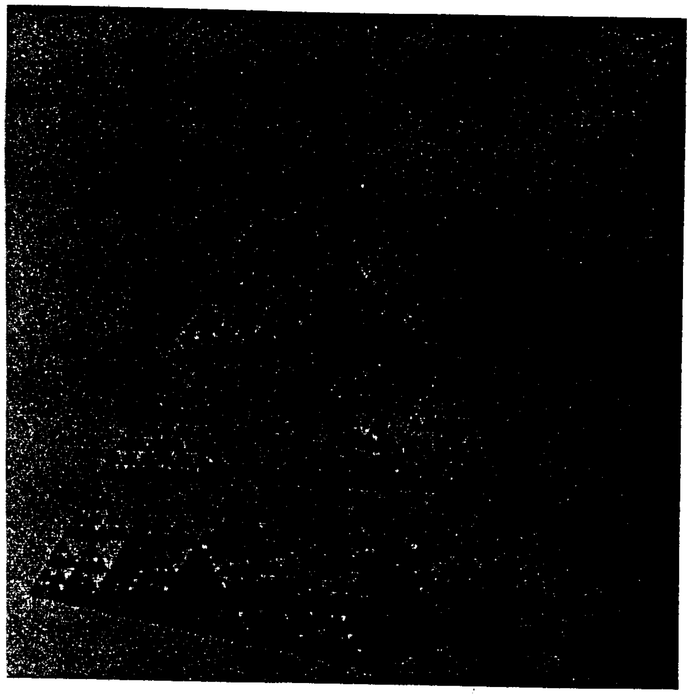
[Vatti 85] Reddy Vatti. Multiprocessor ray tracing. Master's thesis, University of Calgary, Dept. of Computer Science, 1985.

[Wallace 87] John R. Wallace, Michael F. Cohen, and Donald P. Greenberg. A two-pass solution to the rendering equation. In *Computer Graphics*, volume 21. ACM SIGGRAPH, 1987.

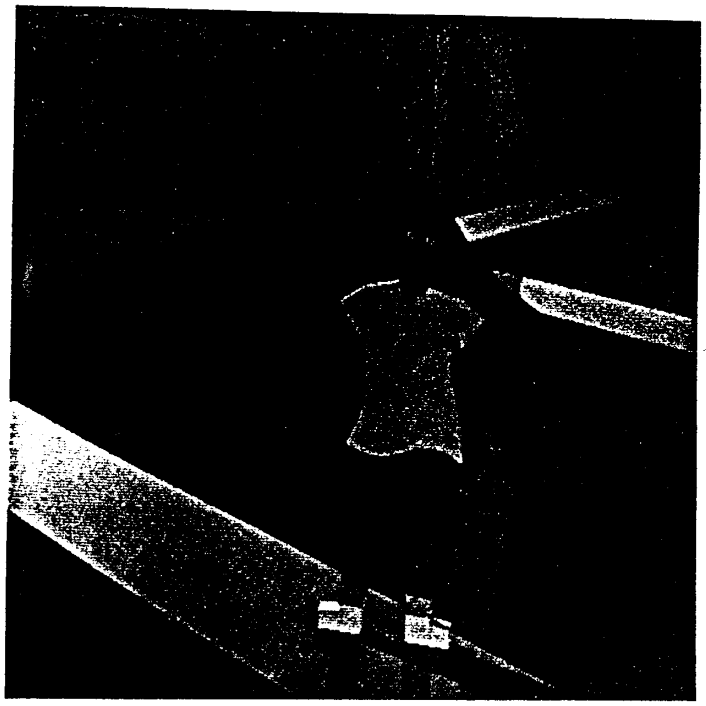
[Wallace 89] John R. Wallace, Kells A. Elmquist, and Eric A. Haines. A ray tracing algorithm for progressive radiosity. In *Computer Graphics*, volume 23. ACM SIGGRAPH, 1989.

[Ward 88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. In *Computer Graphics*, volume 22. ACM SIGGRAPH, 1988.

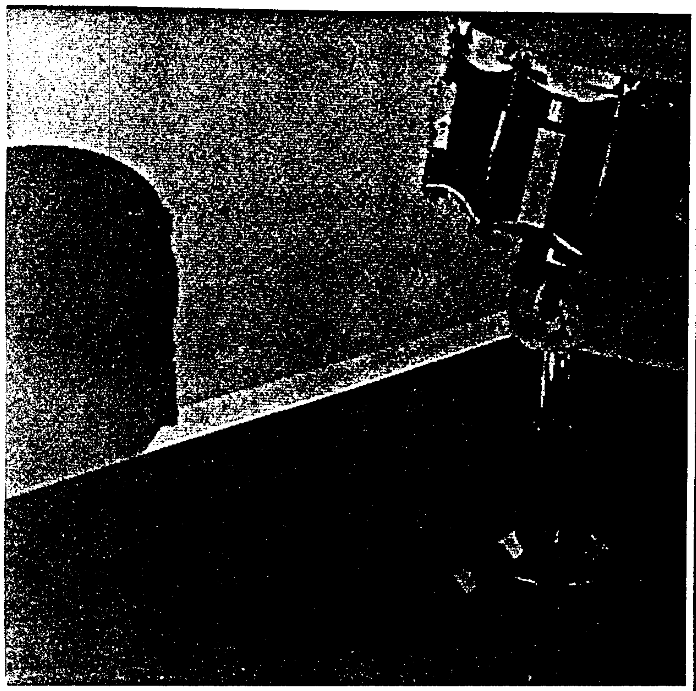
[Whitted 80] Turner Whitted. An improved illumination model for shaded display. *Comm. ACM*, 23(6):343–349, June 1980.



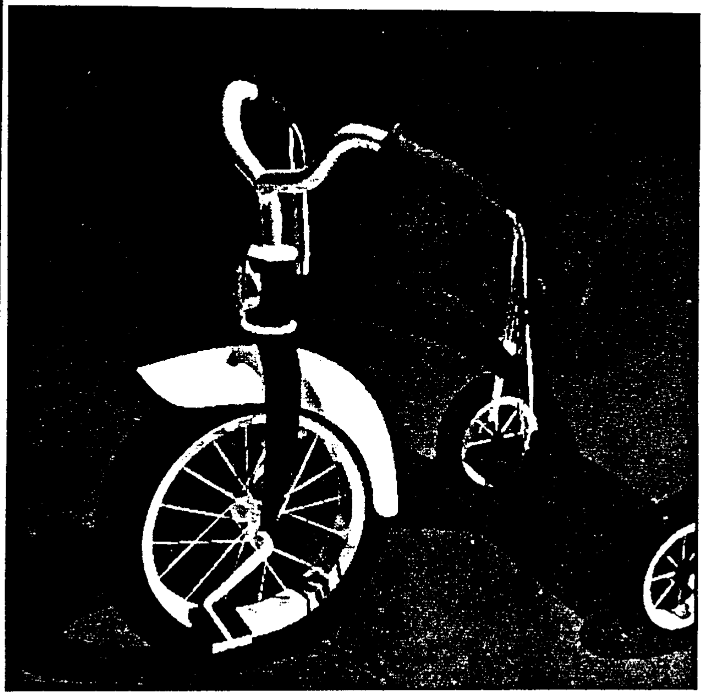
pyramid. 4,096 polygons. 310,000 rays.
10 minutes to render with 1D method.



lumpy. 5,938 polygons. 972,000 rays.
47 minutes to render with 1D method.



drum. 59,672 polygons. 575,000 rays.
120 minutes to render with 1D method.
Model courtesy of Alias Research.



trike. 130,398 polygons. 490,000 rays.
43 minutes to render with 1D method.
Model courtesy of Alias Research.

.

Model courtesy of Alias Research.

120 minutes to render with 1D method
Model courtesy of Alias Research.

odel courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Mo

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Research.

to render with 1D method.
of Alias Research.

Model courtesy of Alias Research.

120 minutes
Model courtesy

Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

search.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Re

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Research.

method.

Model courtesy of Alias Research.

120 minutes to render with 1D
Model courtesy of Alias Research.

Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Research.

minutes to render with 1D method.
l courtesy of Alias Research.

Model courtesy of Alias Research.

120
Mode

Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

f Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy o

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Research.

with 1D method.
arch.

Model courtesy of Alias Research.

120 minutes to render w
Model courtesy of Alias Rese

Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Research.

Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Research.

render with 1D method.
lias Research.

Model courtesy of Alias Research.

120 minutes to
Model courtesy of A

Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

ch.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Resear

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Research.

hod.

Model courtesy of Alias Research.

120 minutes to render with 1D me
Model courtesy of Alias Research.

Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Research.

utes to render with 1D method.
rtesy of Alias Research.

Model courtesy of Alias Research.

120 mi
Model co

Model courtesy of Alias Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

s Research.

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Ali

120 minutes to render with 1D method.
Model courtesy of Alias Research.

Model courtesy of Alias Research.