

THE UNIVERSITY OF CALGARY

# Adaptive Voxel Subdivision for Ray Tracing

BY

David A. J. Jevans

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE  
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

November, 1990

© David A. J. Jevans 1990



National Library  
of Canada

Bibliothèque nationale  
du Canada

Canadian Theses Service    Service des thèses canadiennes

Ottawa, Canada  
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-66971-3

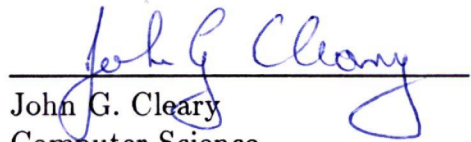
Canada

**THE UNIVERSITY OF CALGARY**  
**FACULTY OF GRADUATE STUDIES**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Adaptive Voxel Subdivision for Ray Tracing," submitted by David A. J. Jevans in partial fulfillment of the requirements for the degree of Master of Science.



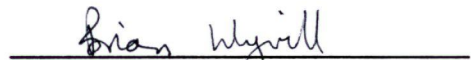
\_\_\_\_\_  
Supervisor Brian R. Gaines  
Computer Science



\_\_\_\_\_  
John G. Cleary  
Computer Science



\_\_\_\_\_  
Donald C. Lawton  
Geology and Geophysics



\_\_\_\_\_  
Brian L. M. Wyvill  
Computer Science

Date November 15, 1990

## Abstract

A new algorithm is presented to reduce the number of ray-object intersection calculations for ray tracing. This algorithm, called adaptive voxel subdivision, uses a hierarchy of uniform three dimensional grids to subdivide a scene. Rays are traced through the grid hierarchy and are tested for intersection with only those objects that lie within the voxels through which they pass. A modified uniform grid traversal algorithm is used to traverse rays through the hierarchical grid structure.

A new tool for examining scene structure, the object distribution graph, is used to examine the distribution of objects in a number of complex scenes. It is seen that the distribution of objects in these scenes is non-uniform. Despite the non-uniformity of object distribution, these scenes exhibit areas of local uniformity, combined with a small number of very dense areas.

The rendering performance of the adaptive voxel subdivision algorithm is compared with octree and uniform subdivision algorithms. The adaptive voxel subdivision algorithm outperforms both the octree and uniform subdivision algorithms when rendering large complex scenes, as it can adapt to areas of high object density, yet uniformly subdivides areas of local uniformity.



## Acknowledgements

As a young, overfunded graduate student, I would often pass idle hours drinking medicinal alcoholic concoctions, and firing shotgun blasts at pieces of paper nailed to the side of the university's orgone accumulator outhouse. Imagine my surprise when I discovered that the performance graphs of my ray tracing algorithms exactly matched the patterns blasted into those pellet ridden sheafs! Seeing this as a sign of divine acceptance, of some kind of grand conjunction of information, I collected my results into the volume of forbidden knowledge that you hold before you.

This scripture, my indoctrination into the secret cult of science, could not have been completed without the enthusiastic support of my supervisor, Brian Gaines. I must also thank Brian Wyvill, the gravity defying sprite who, many years ago, began my apprenticeship of the black art of computer graphics; John Cleary, for his lessons in levitation; and Brian Unger, who funded me during the early stages of my work.

Many thanks to my evil twin, Michael Chmilar, for his constant support and help in the preparation of this manuscript; to Evan Bedford and Chris Bone for the Yaqui way of knowledge; to David Hankinson for the countless trips to Sev; and to the entire graphics group, for their many and varied contributions to the Graphicsland environment.

None of this would have been possible without the support of my parental units, Martin and Gwynneth Jevans. Thank you.

I offer a final word of thanks to Andrea Gail Kelley, who motivated me to complete this thesis in a finite amount of time.

# Contents

<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Statement of Thesis . . . . .	1
1.2 Ray Tracing . . . . .	2
1.2.1 Extending the Illumination Model . . . . .	3
1.2.2 Aliasing . . . . .	3
1.2.3 Other Techniques . . . . .	5
1.3 The Cost of Ray Tracing . . . . .	5
1.4 Acceleration Methods . . . . .	6
1.4.1 Speeding up Ray/Object Intersections . . . . .	6
1.4.2 Exploiting Parallelism . . . . .	7
1.4.3 Reducing the Number of Rays . . . . .	7
1.4.4 Reducing the Number of Ray-Object Intersections . . . . .	10
1.5 Thesis Organisation . . . . .	11
<b>2 Reducing the Number of Ray-Object Intersection Tests</b>	<b>12</b>
2.1 Object Grouping . . . . .	12
2.1.1 Hierarchy Construction . . . . .	14
2.2 Space Subdivision . . . . .	15
2.2.1 Octree Subdivision . . . . .	15
2.2.2 Uniform Subdivision . . . . .	17
2.2.3 Fragmentation . . . . .	19
2.3 Ray Classification . . . . .	20
<b>3 Ray Tracing Complex Scenes</b>	<b>22</b>
3.1 Discussion of Existing Algorithms . . . . .	22
3.1.1 Ray Classification . . . . .	22
3.1.2 Octrees and Bounding Volume Hierarchies . . . . .	23
3.1.3 Uniform Subdivision . . . . .	24
3.2 Measuring Scene Complexity . . . . .	25
3.2.1 Object Distribution Graphs . . . . .	26
3.2.2 Test Scenes . . . . .	27
3.2.3 Observations . . . . .	34
3.3 Adaptive Voxel Subdivision . . . . .	34

<b>4</b>	<b>A Space Subdivision Tool</b>	<b>37</b>
4.1	Details of the Ray Tracer . . . . .	37
4.1.1	Anti-Aliasing . . . . .	38
4.1.2	Renderable Objects . . . . .	38
4.2	Implementation of Ray Traversal . . . . .	39
4.2.1	Horizontal Traversal . . . . .	39
4.2.2	Horizontal Traversal Initialisation . . . . .	42
4.2.3	Vertical Traversal . . . . .	43
4.2.4	Lazy Subdivision . . . . .	45
4.2.5	Grid Data Structure . . . . .	46
<b>5</b>	<b>Algorithm Comparison</b>	<b>47</b>
5.1	Graphing Rendering Performance . . . . .	47
5.1.1	Rendering Details . . . . .	49
5.2	Octree Subdivision . . . . .	49
5.2.1	Choosing the Subdivision Threshold . . . . .	49
5.2.2	Octree Performance . . . . .	51
5.3	Uniform Subdivision . . . . .	56
5.4	Adaptive Voxel Subdivision . . . . .	61
5.4.1	A Subdivision Heuristic . . . . .	61
5.4.2	Performance of Adaptive Voxel Subdivision . . . . .	61
5.4.3	A Revised Heuristic . . . . .	65
5.5	Comparing the Algorithms . . . . .	70
5.5.1	Comparison of Performance for the Cubes scene . . . . .	70
5.5.2	Comparison of Performance for the Tree scene . . . . .	72
5.5.3	Comparison of Performance for the Lumpy scene . . . . .	73
5.5.4	Comparison of Performance for the Drum scene . . . . .	75
5.5.5	Comparison of Performance for the Trike scene . . . . .	77
5.5.6	Comparison of Performance for the Car scene . . . . .	78
5.6	Conclusions . . . . .	80
5.6.1	Adaptive Voxel Subdivision Heuristics . . . . .	80
5.6.2	The Effect of Scene Structure . . . . .	81
5.6.3	Ease of Use . . . . .	82
<b>6</b>	<b>Conclusion</b>	<b>83</b>
6.1	Future Work . . . . .	83
6.1.1	Controlling the Adaptive Voxel Subdivision . . . . .	84
6.1.2	Optimisations to the Adaptive Voxel Subdivision Algorithm . . . . .	84
6.1.3	A Hybrid Algorithm . . . . .	86
6.2	Conclusion . . . . .	87

*CONTENTS*

vii

**Bibliography**

88

## List of Figures

1.1	Ray tracing. . . . .	4
2.1	Hierarchy of bounding volumes. . . . .	13
2.2	Traversal of a hierarchy of bounding volumes. . . . .	14
2.3	Octree. . . . .	16
2.4	Octree traversal. . . . .	17
2.5	Uniform space subdivision. . . . .	18
2.6	Object spans voxels. . . . .	19
2.7	5D ray tracing. . . . .	21
3.1	Continued narrowing does not reduce candidate set. . . . .	23
3.2	Cubes. . . . .	28
3.3	Tree. . . . .	29
3.4	Lumpy. . . . .	30
3.5	Drum. . . . .	31
3.6	Trike. . . . .	32
3.7	Car. . . . .	33
3.8	Hierarchical three dimensional grids. . . . .	35
4.1	Cleary and Wyvill algorithm. . . . .	39
4.2	Pseudocode for the Cleary and Wyvill algorithm. . . . .	40
4.3	Pseudocode for the revised Cleary and Wyvill algorithm. . . . .	41
4.4	Initialising $\Delta x$ . . . . .	43
4.5	Initialising $dx$ . . . . .	43
4.6	Finding $rx_1, ry_1$ . . . . .	44
4.7	Finding $rx_1, ry_1$ . . . . .	44
5.1	Octree performance on the <b>Cubes</b> scene. . . . .	52
5.2	Octree performance on the <b>Tree</b> scene. . . . .	52
5.3	Octree performance on the <b>Lumpy</b> scene. . . . .	53
5.4	Octree performance on the <b>Drum</b> scene. . . . .	53
5.5	Octree performance on the <b>Trike</b> scene. . . . .	54
5.6	Octree performance on the <b>Car</b> scene. . . . .	54
5.7	Over-subdivision of a large object. . . . .	55
5.8	Over-subdivision of small objects. . . . .	56
5.9	Uniform subdivision performance on the <b>Cubes</b> scene. . . . .	57
5.10	Uniform subdivision performance on the <b>Tree</b> scene. . . . .	57
5.11	Uniform subdivision performance on the <b>Lumpy</b> scene. . . . .	58

5.12	Uniform subdivision performance on the <b>Drum</b> scene. . . . .	58
5.13	Uniform subdivision performance on the <b>Trike</b> scene. . . . .	59
5.14	Uniform subdivision performance on the <b>Car</b> scene. . . . .	59
5.15	The effect of increasing subdivision granularity on a large polygon. . .	61
5.16	Adaptive voxel subdivision performance on the <b>Cubes</b> scene. . . . .	62
5.17	Adaptive voxel subdivision performance on the <b>Tree</b> scene. . . . .	62
5.18	Adaptive voxel subdivision performance on the <b>Lumpy</b> scene. . . . .	63
5.19	Adaptive voxel subdivision performance on the <b>Drum</b> scene. . . . .	63
5.20	Adaptive voxel subdivision performance on the <b>Trike</b> scene. . . . .	64
5.21	Adaptive voxel subdivision performance on the <b>Car</b> scene. . . . .	64
5.22	Revised adaptive voxel subdivision performance on the <b>Cubes</b> scene. .	67
5.23	Revised adaptive voxel subdivision performance on the <b>Tree</b> scene. .	67
5.24	Revised adaptive voxel subdivision performance on the <b>Lumpy</b> scene. .	68
5.25	Revised adaptive voxel subdivision performance on the <b>Drum</b> scene. .	68
5.26	Revised adaptive voxel subdivision performance on the <b>Trike</b> scene. .	69
5.27	Revised adaptive voxel subdivision performance on the <b>Car</b> scene. . .	69
5.28	Comparison of algorithm performance for the <b>Cubes</b> scene. . . . .	71
5.29	Comparison of algorithm performance for the <b>Tree</b> scene. . . . .	72
5.30	Comparison of algorithm performance for the <b>Lumpy</b> scene. . . . .	74
5.31	Comparison of algorithm performance for the <b>Drum</b> scene. . . . .	76
5.32	Comparison of algorithm performance for the <b>Trike</b> scene. . . . .	77
5.33	Comparison of algorithm performance for the <b>Car</b> scene. . . . .	79

# Chapter 1

## Introduction

### 1.1 Statement of Thesis

Methods for reducing the number of ray-object intersection tests required to ray trace an image are investigated. Voxel subdivision is argued to be the most appropriate method for complex scenes. A new type of voxel based spatial subdivision is developed to deal with scenes containing areas of varying complexity. The new method is a combination of uniform subdivision and octree techniques.

A tool that combines fast grid traversal and hierarchical subdivision is used to implement octree, uniform, and the new adaptive voxel subdivision schemes. The subdivision parameters of each of these three algorithms are varied to characterize their performance and to find the fastest rendering time for a number of complex scenes. These times are compared and the following conclusions about the adaptive voxel subdivision algorithm are reached:

- The algorithm has smaller subdivision tree depths than octrees, requiring fewer vertical traversals to trace a ray through a scene.
- The algorithm has more adaptivity to varying scene complexity than uniform subdivision.
- The algorithm has the fastest rendering times for complex scenes.

The main achievements presented in this thesis are:

- A ray tracing tool that implements octree, uniform, and adaptive voxel subdivision.
- Subdivision heuristics for adaptive voxel subdivision.
- Use of object distribution graphs to analyse the structure of a number of complex scenes.
- A comparison of the performance of octree, uniform, and adaptive voxel subdivision algorithms when rendering complex scenes.
- Justification for the adaptive voxel subdivision algorithm based on empirical results and previous theoretical analysis.

## 1.2 Ray Tracing

*Ray tracing* was first introduced by Appel [Appel 68] and MAGI [MAGI 68] to render engineering designs. Ray tracing did not enjoy widespread use until Whitted [Whitted 80] and Kay [Kay 79] incorporated it into a general illumination model.

Object descriptions and light sources are placed into a single three dimensional *object space*. An *eye point* and *virtual viewing screen* are placed in the object space and an image, the projection of the scene onto the viewing screen, is calculated.

Simple ray tracing involves *firing* one ray per pixel from the eye point through the virtual screen and into the scene. A ray is tested for intersection with all objects in the scene. Successful intersections are sorted by distance along the ray from its



origin at the virtual viewing screen. The closest intersection to the ray's origin is the object which is visible from the pixel. A reflectance model is used to shade the closest intersection point, and the resulting color is assigned to the pixel.

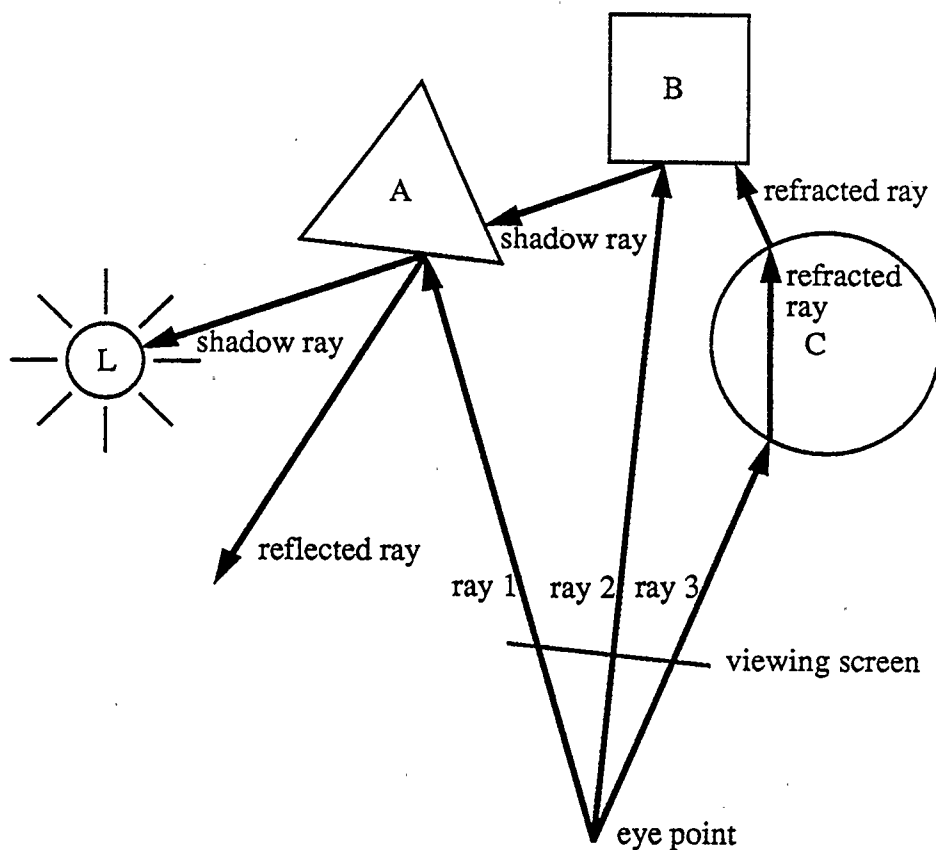
### 1.2.1 Extending the Illumination Model

Shadows may be incorporated into this illumination model by firing rays towards light sources from visible intersection points. If a *shadow ray* intersects an object before the light source, then it obscures the light and prevents it from illuminating the visible surface.

A further extension of the algorithm is to fire reflected and refracted rays if a surface's attributes so warrant. These rays are fired through a recursive application of the ray tracing algorithm, and their results are incorporated into the pixel's color value. This process continues until a predefined *shade tree* depth is reached, or the secondary ray's contribution to the pixel's color falls below a minimal threshold [Glassner 89]. This revised algorithm is what is normally referred to as "ray tracing" (figure 1.1).

### 1.2.2 Aliasing

*Aliasing*, the result of a low frequency sample of a high frequency signal, is apparent in ray tracing due to its point sampling nature [Dippé 85] [Heckbert 89]. Aliasing usually manifests itself as jagged lines along the edges of visible objects or textures. *Super-sampling*, firing more than one ray per pixel and filtering the results, replaces aliasing with noise, a much less noticeable artifact. Super-sampling is usually done only in areas where large changes in pixel intensity or color are detected, and may



Ray 1 intersects object A, which is reflective.  
 A shadow ray is send towards the light source, L, and a reflected ray is fired.  
 Ray 2 hits object B, which is in shadow from object A.  
 Ray 3 hits transparent object C, and refracted rays are fired.

Figure 1.1: Ray tracing.

require as many as sixteen rays per pixel [Mitchell 87].

### 1.2.3 Other Techniques

Additional phenomena such as *penumbra*, *diffuse reflections*, *translucency*, and *motion blur* can be modelled using a technique known as *distributed ray tracing* [Cook 84]. *Stochastic sampling*, sampling using probability distributions, may be used to obtain more realistic images from the ray tracing method [Kajiya 86]. More recently, ray tracing has been used in conjunction with *radiosity* techniques, which model the interaction of light between diffusely reflecting surfaces, to create photo-realistic images [Wallace 87] [Ward 88] [Wallace 89] [Sillion 89] [Heckbert 90].

## 1.3 The Cost of Ray Tracing

The naive ray tracing algorithm tests each ray for intersection with every object in the scene, and finds the closest intersection to the ray's origin. This rapidly becomes impractical as scene complexity increases to hundreds of thousands of objects, and as image size and anti-aliasing quality increase. Kingdon [Kingdon 86] performs an analysis of the naive algorithm and shows that for even simple scenes, image generation may require several days.

Kingdon estimates the cost of rendering a single image to be:

$$Cost = XYN(1 + pk) \frac{1 - (pq)^{b+1}}{1 - pq} \quad (1.1)$$

where the image is of size  $X$  by  $Y$ ,  $N$  is the number of objects in the scene,  $p$  is the probability of a ray intersecting any object,  $q = q_s + q_t$  is the sum of the probabilities that the intersected object is reflective and refractive,  $k$  is the number

of light sources, and  $b$  is the maximum number of recursive “bounces”, or maximum tree depth, allowed.

Evaluating formula 1.1 for some “typical” values gives an idea of the cost of naive ray tracing. Let  $X = 512$ ,  $Y = 512$ ,  $p = 0.5$ ,  $q = 0.5$ ,  $k = 2$ , and  $b = 3$ . This results in an estimate of the rendering cost being  $6.96 * 10^5 N$  ray-object intersection tests. The ray tracing implementation used in this thesis has a ray-object intersection calculation time of  $27.5us$  (see section 5.2.1 in chapter 5). Using this figure, Kingdon’s equation estimates that an image of a scene consisting of a single object requires 19.1 seconds to render. A scene containing 1,000 objects requires 5.3 hours to render. This time increases to 53.1 hours to render 10,000 objects, and to 531 hours to render 100,000 objects. This makes the rendering of complex scenes impractical with a naive ray tracing algorithm.

## 1.4 Acceleration Methods

While ray tracing in its naive form is an expensive operation, there are many techniques available to improve its performance. This section examines a number of directions for improving the performance of the basic ray tracing algorithm.

### 1.4.1 Speeding up Ray/Object Intersections

Whitted [Whitted 80] noted that the ray-object intersection process may be sped up by surrounding objects with simple bounding volumes, with which rays may be more quickly intersected than with the actual objects. Rays which do not intersect a bounding volume need not be checked for intersection with the object within. This

technique was also used by Roth [Roth 82] to render solid models.

### 1.4.2 Exploiting Parallelism

Since the calculation of the color at each pixel is independent from the calculation of other pixels' colors, a parallel implementation suggests itself. The most straightforward technique is to duplicate the entire object space on a number of processors, and direct each processor to render a portion of the final image [Nishimura 83]. More effective use of distributed memory can be had by scattering objects among processors, and performing intersection calculations in parallel for each ray.

More recent algorithms partition space into disjoint volumes, and distribute these volumes and the objects that lie inside them among processors. As rays are traversed through the scene they are passed between processors which perform ray-object intersection calculations for the objects in their volumes [Dippé 84] [Vatti 85] [Cleary 86] [Kobayashi 87] [Scherson 88]. The object space partitions may require adjustment during rendering to balance computational load.

This thesis deals with single-processor algorithms, but these can often be adapted to parallel architectures [Nemoto 86] [Pearce 87] [Jevans 89b].

### 1.4.3 Reducing the Number of Rays

A reduction in the number of rays required to render an image can be had by combining ray tracing with other rendering methods. A *Z-buffer* algorithm [Sutherland 74] can be used to determine the visible surface at each pixel, and ray tracing used only to cast shadows and calculate secondary illumination from reflective and refractive surfaces [Weghorst 84]. The number of secondary rays can be reduced by

firing them only if their contribution to the total intensity of the pixel is significant [Hall 83] [Arvo 90].

### Shadow Testing Accelerators

Shadow ray intersection calculations can be reduced by using *light buffers*, *voxel occlusion testing*, or *shadow object caches*.

A light buffer [Haines 86] is a cube, aligned with the scene's coordinate axes, that surrounds a light source. The faces of a light buffer are subdivided into a number of viewing frustums. Each frustum contains a list of the objects that are visible to the light through it. To trace a shadow ray toward a light source, the viewing frustum of its light buffer through which the ray will pass is determined, and the ray is tested for intersection with only the objects in the frustum's list.

Voxel occlusion testing [Woo 90] is used in spatial subdivision algorithms to reduce the distance that shadow rays must be traced. In a pre-processing step, each volume of space is checked to see if it is completely, partially, or not occluded from each light source. When tracing a shadow ray toward a light source, the occlusion status of each volume through which it passes is checked. If the shadow ray enters a volume that is not occluded from the light, then the ray will not be occluded, and tracing can stop. Similarly, if the volume is completely occluded from the light source, then the shadow ray will also be occluded, and tracing can stop. If the voxel is partially occluded from the light source, then tracing of the shadow ray must continue.

Shadow caches [Haines 86] make use of the ray coherence property that adjacent rays will most likely be shadowed by the same object. A shadow cache is a pointer,

stored with each light source, to the last object that cast a shadow from the light. Before tracing a shadow ray toward a light, it is tested for intersection with the object pointed to by the light's shadow cache. If the shadow ray intersects that object, then a shadow is cast, and tracing is not required. If the shadow ray does not intersect that object, then it is traced toward the light in the usual fashion. If the shadow ray reaches the light without hitting an object, then the shadow cache is cleared. Light source shadow caches are cleared at the end of every scanline, since it is unlikely that the same object casts shadows that are visible to pixels at opposite ends of a scanline.

### Ray Coherence

Speer, DeRose, and Barsky [Speer 85] present a technique for exploiting ray-to-ray coherence when ray tracing. They construct a cylindrical bounding volume around a ray with the radius being the distance to the nearest non-intersecting object. Successive rays, if they travel inside the previous ray's cylinder, need only be tested against the objects with which the previous ray intersected. The method degrades rapidly as the number of objects increases.

Hanrahan [Hanrahan 86] extends Speer's algorithm with a revised bounding ray volume and a breadth-first rendering technique. As the number of objects increases, the algorithm provides significantly better performance than Speer's method, but the paper presents few results, and there has been little in the way of follow-up research.

Joy [Joy 86] used ray coherence to reduce initialization of newton iteration parameters, which he used to intersect rays with parametric patches.

The problem with ray-to-ray coherence algorithms is that they demand a high

degree of similarity between adjacent intersection trees to obtain any performance gain. Highly complex scenes may have a different visible surface at each pixel, with little similarity between ray intersection trees. Distributed ray tracing methods, which randomly perturb ray directions, reduce ray-to-ray coherency, limiting the usefulness of these techniques. Radiosity algorithms which use ray tracing to find *form factors*, the area of each surface which is visible from every other surface, may have little if any ray-to-ray coherence, necessitating the use of some other type of speedup technique.

### Other Methods

There are a number of other methods, similar to ray tracing, that trace clusters of rays or sweep areas through space. *Beam tracing* [Heckbert 84], sweeps areas of space through a scene to form “beams”. The first beam to be traced is the viewing screen, and subsequent beams are traced when objects are intersected or reflections are required. *Cone tracing* [Amanatides 84] traces cones of light rather than rays, enabling effects such as penumbrae and dull reflections. Firing a single cone per pixel can provide better anti-aliasing than tracing a large number of rays through each pixel and filtering the results. *Pencil tracing* [Shinya 87], a similar technique, traces clusters of rays by firing a single ray for each cluster and using paraxial approximation theory to approximate the paths of the untraced rays in a cluster.

#### 1.4.4 Reducing the Number of Ray–Object Intersections

This thesis is concerned with accelerating the ray tracing process by reducing the number of ray–object intersection calculations performed per ray. Since a ray travels



through a small subset of the object space, a means of culling objects which are not near its path is desirable. Object culling methods can be classified as object grouping, space subdividing, or ray classifying. Chapter 2 presents a survey of these methods.

## 1.5 Thesis Organisation

A review of techniques for reducing the number of ray-object intersection tests required to trace a ray through a scene is presented in chapter 2.

Chapter 3 discusses the algorithms reviewed in chapter 2 with regard to their performance on complex scenes. By unifying the results of previous theoretical and empirical analyses, voxel subdivision is argued to be the superior scheme. A number of scenes are examined to determine the uniformity of object distribution and object size. It is shown that these scenes do not conform to the assumptions made by proponents of uniform subdivision. This leads to the development of a new technique that combines octree and uniform voxel subdivision.

Chapter 4 describes a tool for implementing octree, uniform, and the new adaptive voxel subdivision algorithms.

The performance of the octree, uniform, and adaptive voxel subdivision algorithms is plotted and compared in chapter 5.

In chapter 6, the results are summarised, and directions for future work are proposed.

## Chapter 2

# Reducing the Number of Ray–Object Intersection Tests

Ray tracing can be made faster by reducing the number of ray–object intersection calculations performed per ray. Since a ray travels through a small subset of the object space, a means of culling objects which are not near its path is desirable. Object culling methods can be classified as object grouping, space subdividing, or ray classifying.

### 2.1 Object Grouping

Rubin [Rubin 80] determined that the majority of time in ray tracing was spent performing ray–object intersection calculations. Rubin and Whitted [Rubin 80] proposed the idea of using a hierarchy of bounding boxes to group objects that have some spatial locality. The hierarchy is built in a bottom up fashion, until the entire scene is contained by a single bounding box. Figure 2.1 is a two dimensional example of a hierarchy of bounding volumes and its data structure. For the sake of clarity, the bounding boxes are not tight-fitting around the objects.

When a ray is traced, it is tested for intersection with the root bounding box. If it intersects the box, then it must be checked for intersection with the root’s children. This process continues recursively until the ray has been tested for intersection with the objects that lie inside the leaf volumes with which it intersects. These intersec-

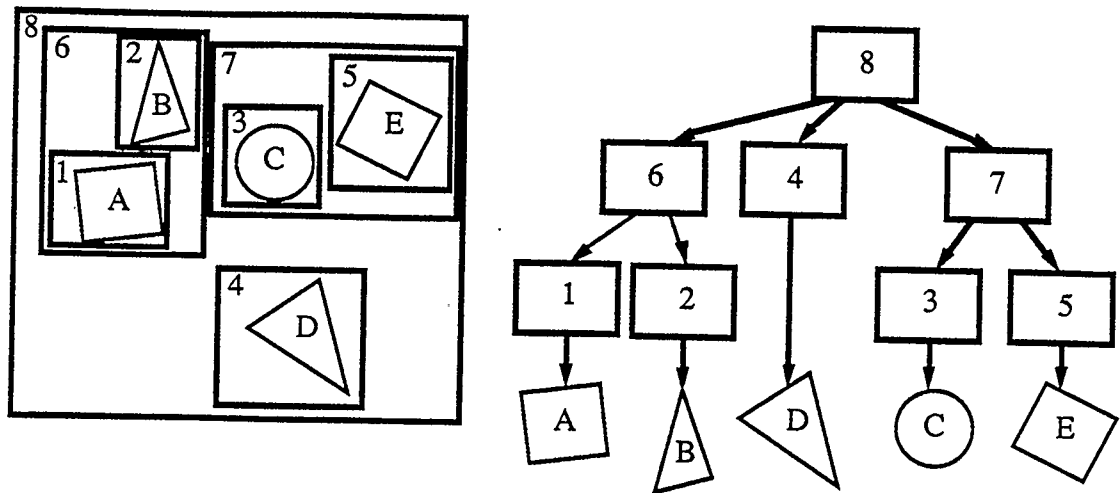


Figure 2.1: Hierarchy of bounding volumes.

tions are sorted by distance along the ray to determine the closest intersection point. Figure 2.2 illustrates a ray traversing a bounding volume hierarchy. The bounding volumes and objects that are tested for intersection are shaded.

Kay and Kajiya [Kay 86] refined the idea of bounding box hierarchies to include arbitrarily tight fitting bounding volumes. Their method also sorts child volume intersections by distance along the ray, and processes them in that order, as was done by Kajiya to render fractals [Kajiya 83]. Processing of a ray stops when a ray-object intersection is found that is closer to the ray's origin than the intersection of the ray with the next child volume. This can significantly reduce the number of intersection calculations required to trace a ray.

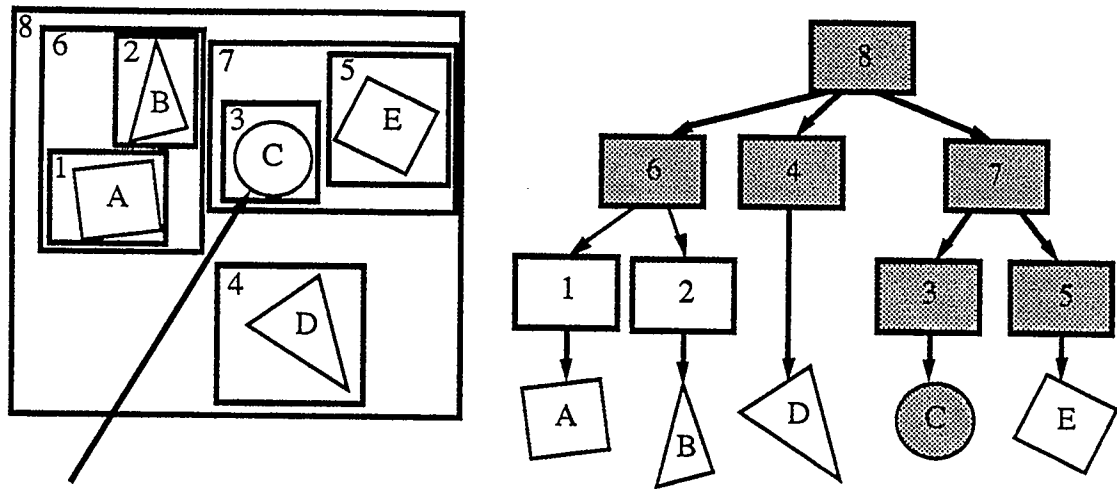


Figure 2.2: Traversal of a hierarchy of bounding volumes.

### 2.1.1 Hierarchy Construction

The performance of hierarchical bounding volume algorithms depends largely on the construction of the hierarchies. If a hierarchy is poorly constructed, with little attention to the locality of the bounding volumes as they are grouped together, it may be necessary to intersect a ray with every leaf volume in the tree. Another drawback to the hierarchical bounding volume method is that the hierarchies are constructed manually by the user, often requiring several days of work.

An algorithm for automatically constructing bounding volume hierarchies was proposed by Goldsmith and Salmon [Goldsmith 87]. Their method builds a hierarchy in a top-down fashion using a heuristic tree search. A new object is inserted into the tree at a location that will result in the minimal surface area of the resulting parent bounding volume. Another approach, suggested by Kay and Kajiya [Kay 86] and discussed in detail by Scherson and Caspary [Scherson 87], uses a  $k$ -d tree or octree, described in section 2.2.1, to guide the creation of the object hierarchy.

## 2.2 Space Subdivision

An alternative to object grouping is the space subdivision technique. Space is subdivided into discrete volumes, and each volume has a list of the objects that lie wholly or partially inside it. Rays traverse the subdivided space, moving from volume to volume, and are tested for intersection only with objects that lie inside the volumes through which they pass. When an intersection of a ray with an object in a volume is found, traversal of the ray stops.

A common form of space subdivision is *voxel* subdivision, where space is subdivided into rectangular prisms aligned with the x, y, z axes of the scene. The advantage of voxel subdivision is that efficient ray traversal algorithms are possible, due to the regular shape of the voxels and their alignment with the primary axes. The two main voxel methods used to accelerate ray tracing are octree and uniform subdivision.

### 2.2.1 Octree Subdivision

Glassner [Glassner 84] used an *octree* [Samet 84], a recursive subdivision of space into eight equal subvolumes, to hierarchically subdivide a scene. The scene is surrounded with a bounding box that serves as the root voxel. If the number of objects in the root voxel is above some pre-defined threshold, the voxel is subdivided into eight equal sized sub-voxels, and the objects are inserted into the sub-voxels. This continues recursively until there are fewer than the pre-defined threshold number of objects in each leaf voxel, or a maximum tree depth is reached. Figure 2.3 illustrates a quadtree, the two dimensional analog of the octree. Note that empty voxels are not

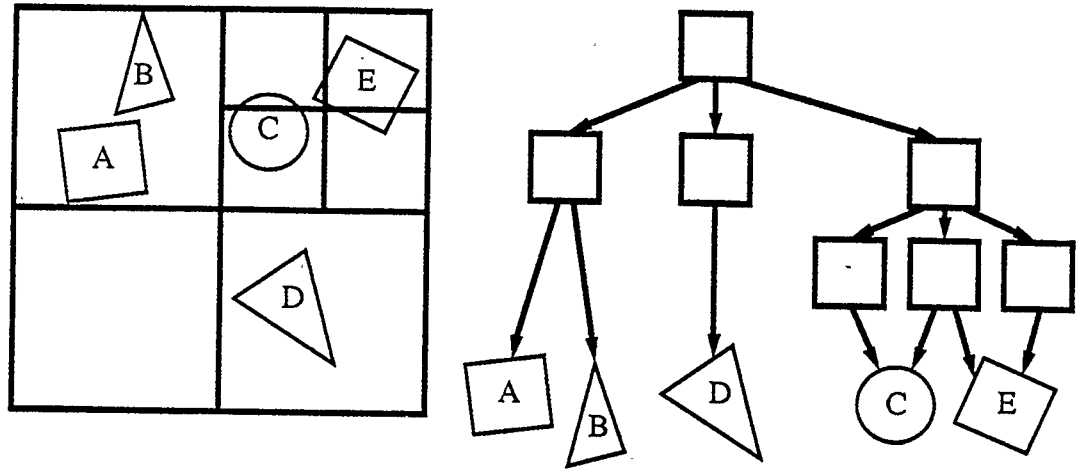


Figure 2.3: Octree.

stored in the data structure.

To trace a ray through the scene, it is first tested for intersection with the root voxel. If the ray intersects the root, and the root is subdivided, the intersection point of the ray and the voxel is used to determine the sub-voxel in which to continue processing. If the sub-voxel is a leaf node and contains objects, the ray is tested for intersection with the objects that lie inside it. If the sub-voxel contains a sub-grid, the algorithm recurses to traverse the ray through it.

Once a sub-voxel has been processed, and if the ray has not intersected an object, the ray must resume its traversal through the grid. The next sub-voxel in the ray's path is determined, and the process repeats until an intersection is found, or the ray exits the bounds of the root voxel. Figure 2.4 illustrates the order in which voxels and objects are visited by a ray.

The octree method has been used to render objects modelled by constructive

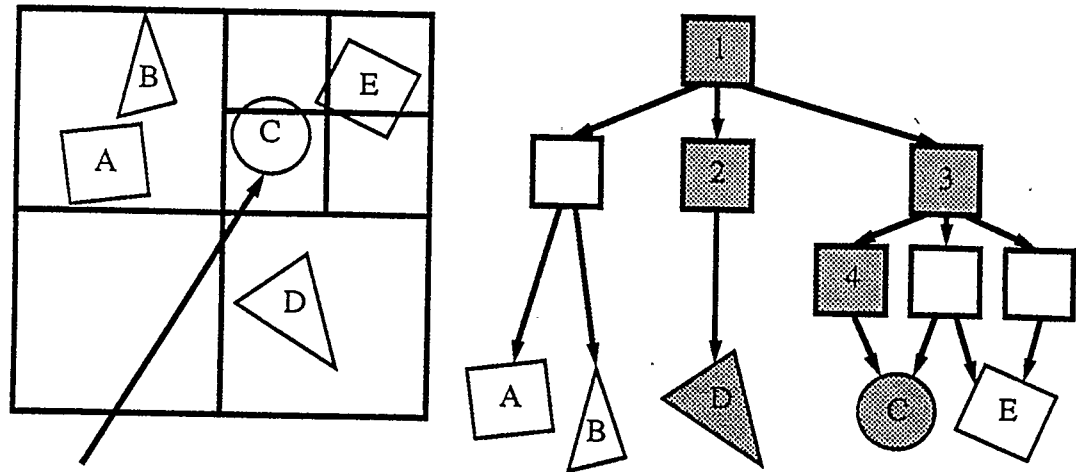


Figure 2.4: Octree traversal.

solid geometry [Wyvill 85] [Kunii 85] [Wyvill 86c] as well as objects modelled with polygons and parametric surfaces.

### *k*-d Trees

Kaplan [Kaplan 85] developed an algorithm, similar to Glassner's octree algorithm, that uses a *k*-d tree [Samet 84] to recursively split a voxel in half along one of the *x*, *y*, or *z* axes at each node. MacDonald and Booth [MacDonald 89] elaborate on the technique and propose a set of heuristics for choosing the axis of subdivision and the position of the splitting plane inside each node.

#### 2.2.2 Uniform Subdivision

A drawback of the octree method is the number of vertical tree traversals required to traverse a ray through the scene. Fujimoto [Fujimoto 86] proposed the use of uniform voxel subdivision, where the object space is subdivided by a regular three

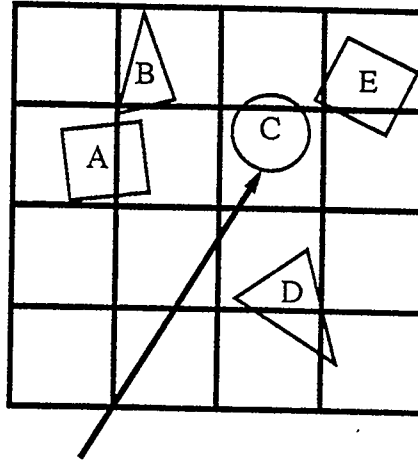


Figure 2.5: Uniform space subdivision.

dimensional grid of voxels (figure 2.5). He developed an iterative algorithm, similar to a two dimensional line drawing algorithm [Foley 90](pages 72-81), for quickly traversing a ray through the voxel grid.

To compare uniform subdivision with octree subdivision, Fujimoto extended his uniform grid traversal algorithm to an octree traversal algorithm. The uniform grid traversal algorithm is used to traverse a ray through the sub-voxel grid at each level of the octree. When a ray encounters a sub-voxel that is itself subdivided, the traversal algorithm “descends” into the sub-voxel, and the ray is traversed through the sub-grid.

Fujimoto compared the performance of the uniform subdivision and octree algorithms on a number of scenes and determined that the uniform subdivision algorithm was superior to the octree algorithm, due to its lack of vertical tree traversals. The validity of this claim will be explored in later chapters.

Optimised uniform grid traversal algorithms have been developed by Cleary and Wyvill [Cleary 88], Amanatides [Amanatides 87], and Snyder and Barr [Snyder 87].



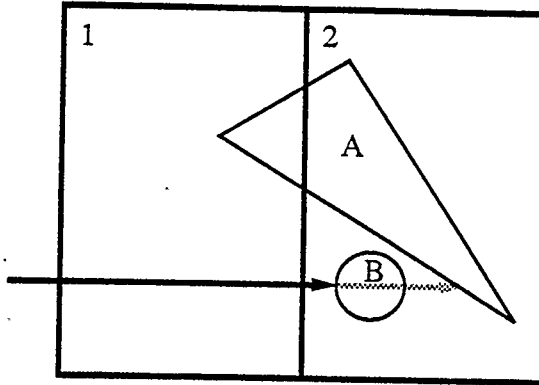


Figure 2.6: Object spans voxels.

### 2.2.3 Fragmentation

Unlike hierarchies of bounding volumes, an object may lie in more than one voxel. This fragmentation has two drawbacks, the first being that ray-object intersection points must be checked to ensure that they lie inside the current voxel. In figure 2.6, the ray is tested for intersection with object A in voxel 1. An intersection is found, but it lies outside of voxel 1, so traversal of the ray must continue. In voxel 2 the ray is tested for intersection with objects A and B, and is found to hit object B before object A.

The second drawback of voxel subdivision is that rays may have to be tested for intersection more than once with an object that spans several voxels (for example, object A in figure 2.6). Duplication of intersection tests can be eliminated through the use of *ray signatures* [Arnaldi 87] [Amanatides 87]. Each ray is assigned a unique number, called a signature. When a ray is tested for intersection with an object for the first time, the result of the test and the ray's signature are stored with the object. Subsequent intersection calculations are avoided by comparing the ray's signature with the one stored in the object being tested against. If they match, then the

intersection results stored with the object are retrieved, and a duplicate intersection calculation need not be performed.

## 2.3 Ray Classification

Arvo and Kirk [Arvo 87] presented a method which combines  $k$ -d tree subdivision and directional volumes to reduce the number of ray-object intersection calculations. Every node in the  $k$ -d tree has a directional volume for each of the six faces of the voxel. Each of these six bounding *hypercubes* contains a list of all objects that could be intersected by a ray originating in the voxel and passing through the face. Rays are represented by a three dimensional origin  $(x, y, z)$  and two spherical angles  $(u, v)$ . Figure 2.7 shows an example of a two dimensional hypercube.

A subdivision hierarchy is created, with each node being split in half along one of the  $x, y, z, u$  or  $v$  axes. A ray is traced by descending the tree to the leaf node that encloses the ray's five dimensional position. This leaf hypercube will contain a list of objects with which the ray may potentially intersect. The ray must be tested for intersection with these objects, and the successful intersection points sorted by distance along the ray to find the closest.

Ohta and Maekawa [Ohta 87] presented a similar algorithm that uses the properties of ray coherence. Every location in the scene that can be a source of rays, such as the eye point and reflective and refractive objects, is bounded by a sphere or convex hull. Each of these bounding volumes is subdivided into a number of viewing direction intervals. For each viewing direction interval, a bounding volume keeps a list of the objects in the scene that are visible to it in that direction. These

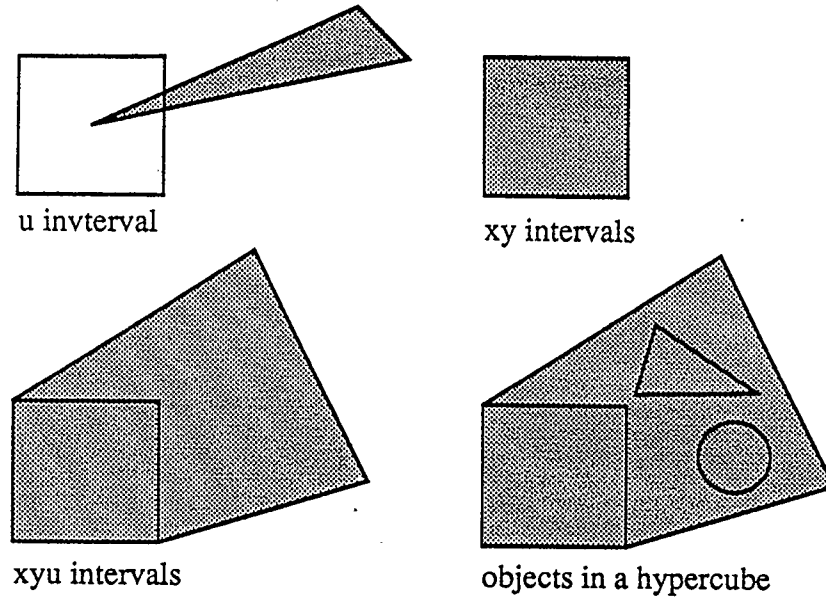


Figure 2.7: 5D ray tracing.

lists are created using ray coherence theorems presented in their paper. Rays that originate in one of these bounding volumes need only be tested for intersection with the objects that are visible in the viewing direction interval that corresponds to the ray's direction.

The authors claim that this algorithm results in constant time ray tracing, independent of the number of objects in the scene. This claim is misleading, however, because the rendering times do not include pre-processing time. For complex environments with many reflective objects, or when shadows are being calculated, these visibility computations must be made for every object. This results in huge pre-processing times, possibly negating the effects of the algorithm.

## Chapter 3

### Ray Tracing Complex Scenes

Much research in computer graphics is targeted towards the generation of photo-realistic images. This involves high quality rendering and detailed modelling. Complex models can take a variety of forms, including polygonal meshes [Allan 89], quadric surfaces [Blinn 86], cubic splines [Bartels 87], and soft objects [Wyvill 86b]. If polygonized [Wyvill 86d] [VonHerzen 87], the resulting scene descriptions are extremely large. Additional complexity arises when detailed models are placed into larger, less complex scenes. This is typical of computer animation, which often involves detailed focal objects amid backgrounds that, as in traditional film making, are simple façades.

#### 3.1 Discussion of Existing Algorithms

This section discusses the algorithms surveyed in chapter 2 with emphasis on their performance when rendering complex scenes.

##### 3.1.1 Ray Classification

The 5D algorithm of Arvo and Kirk [Arvo 87], while performing well on the scenes presented in their paper, may not be well suited to rendering complex scenes due to its large space requirements. Because the algorithm's space requirements depend not only on the number of objects, but on the visibility of the rest of the scene from

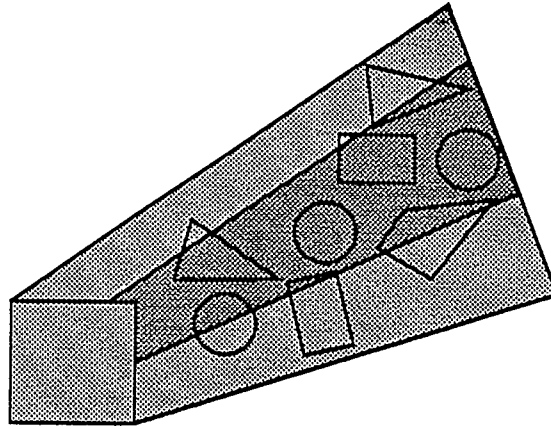


Figure 3.1: Continued narrowing does not reduce candidate set.

each object, its space requirements are  $O(N^2)$ , where  $N$  is the number of objects in the scene. Another problem with the algorithm is that it may perform poorly when complex objects occupy a small viewing volume, as continued recursive narrowing of the hypercube may not reduce the number of objects in it (see figure 3.1).

### 3.1.2 Octrees and Bounding Volume Hierarchies

Scherson and Caspary [Scherson 87] compared octree subdivision and bounding volume hierarchies. They discovered that the number of ray-voxel intersections required to trace a ray through an octree was smaller than the number of ray-volume intersections required to trace a ray through a hierarchy of bounding volumes. This is offset by the cost of duplicating ray-object intersection calculations for objects that lie in more than one leaf voxel of an octree.

To resolve this dilemma they developed a hybrid algorithm that uses octree subdivision to coarsely subdivide a scene, and hierarchies of bounding volumes to order the objects inside the octree leaf nodes. This algorithm outperforms both the octree and bounding volume hierarchy algorithms.

Amanatides and Woo [Amanatides 87] and Arnaldi [Arnaldi 87] concurrently presented the ray signature technique to avoid duplicating ray-object intersection tests in spatial subdivision algorithms (see section 2.2.3, chapter 2). This technique virtually eliminates the overhead of fragmentation, and obviates the need for bounding volume hierarchies inside voxel structures.

### 3.1.3 Uniform Subdivision

Cleary and Wyvill [Cleary 88] presented and analysed a uniform voxel subdivision algorithm. They developed a formula, balancing the time to perform a ray-polygon intersection test and the time to traverse a ray through a voxel, to determine the optimal subdivision *granularity*, the number of voxels that subdivide a scene, thereby minimising ray tracing execution time. Their analysis shows that ray tracing time can be reduced to about twice the irreducible time of initialising each ray, and performing a single intersection calculation and shading for each ray that hits a polygon. Based on this result, Cleary and Wyvill stated that dramatic gains cannot be expected from incremental improvements to ray tracing itself, and that other techniques, such as ray coherence, will be necessary to make further improvements in execution time.

The Cleary and Wyvill analysis of uniform subdivision involves the mean polygon area and perimeter, and assumes that polygons are distributed uniformly throughout the scene. They do not consider the possibility that polygon area or perimeter may vary greatly, or that the distribution of objects in the scene may be non-uniform. In such cases, the calculated optimal subdivision granularity may be optimal for only a subset of the scene.

The following section measures the density of polygons, mean polygon area, mean

polygon perimeter, and the standard deviation of these values for a number of complex scenes. If the standard deviation of these values is high, then the assumptions necessary to Cleary and Wyvill's statement are not supported, and significant improvements to the ray tracing algorithm may still be possible.

### 3.2 Measuring Scene Complexity

Scherson and Caspary [Scherson 87], in their comparison of octree and bounding volume hierarchy methods, examined the distribution of objects for a number of scenes. They subdivided each scene with a uniform grid of 1000 voxels, 10 on a side, and counted the number of polygons in each voxel. The mean number of polygons per voxel and the standard deviation were computed for each scene.

Scherson and Caspary calculated and normalized the standard deviation of the number of objects per voxel with respect to the mean number of objects in *non-empty* voxels. This results in a measure of the uniformity of object distribution in non-empty areas of a scene, not the entire scene. For example, if all objects in a scene lie in a single voxel, thus occupying 1/1000 of the volume of the scene, their measurement technique will record a normalized standard deviation of 0, indicating a uniform distribution. This is highly misleading, as the scene is far from uniform.

To obtain a meaningful measure of object distribution, the standard deviation must be calculated and measured with respect to the number of objects in *all* voxels. The formulation of the standard deviation is given in equation 3.1.

$$\sigma = \sqrt{\frac{\sum (x - \bar{x})^2}{g}} \quad (3.1)$$

where:

- $N$  = the number of objects in the scene
- $g$  = the number of voxels ( $g = 1000$ )
- $x$  = the number of objects in a voxel
- $\bar{x}$  = the average number of objects per voxel ( $\bar{x} = N/g$ )
- $\sigma$  = the standard deviation

The following section uses formula 3.1 to measure object distribution for a number of scenes. The mean and standard deviation of the perimeters and areas of the objects are also measured. The standard deviations are divided by the means to normalize them.

### 3.2.1 Object Distribution Graphs

To visualise the distribution of object densities, an *object distribution graph* is plotted for each scene. The number of objects in a voxel is plotted logarithmically along the horizontal axis, and the percentage of voxels containing a given number of objects is plotted along the vertical axis.

An object distribution graph is useful in identifying areas of local uniformity in scenes that have a large standard deviation of object density. An object density that occurs in a large percentage of voxels, and thus is plotted high on the vertical axis, is an indication of local uniformity. Another indication is a cluster of points on the graph, indicating a number of voxels with similar object densities.

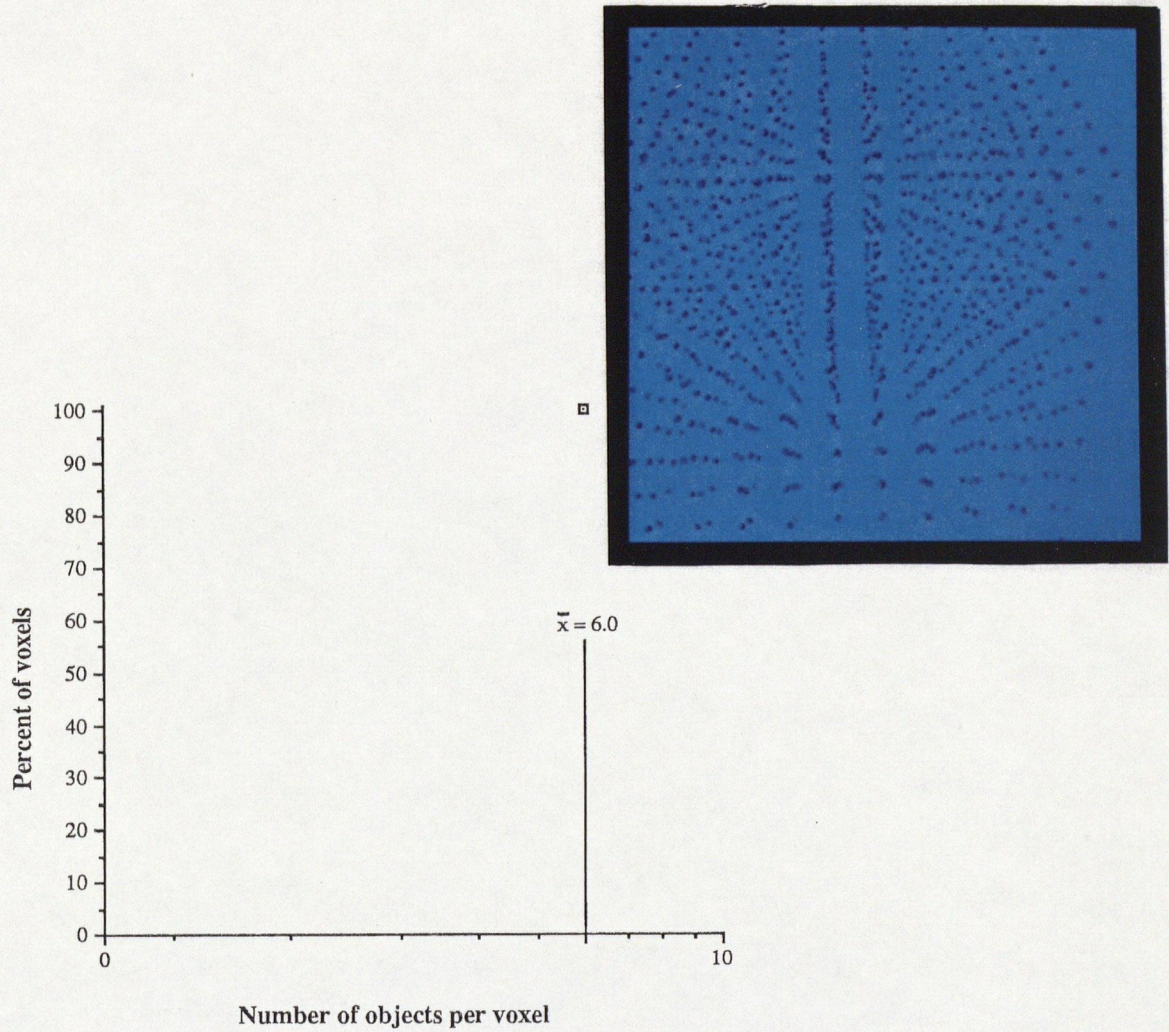


### 3.2.2 Test Scenes

The largest of the scenes that Scherson and Caspary analysed was composed of less than 20,000 polygons. Their scenes were atypical of those commonly rendered, as they consisted of isolated objects with no background. The scenes that will be examined in this section include several test scenes, similar to those used by Scherson and Caspary and Cleary and Wyvill, and a number of scenes taken from animated films and commercial applications. The number of polygons in these scenes ranges from 6,000 to almost 200,000.

The test scenes are composed entirely of planar polygons. They were modelled either with a polygon based modelling system, SFPG, developed at the University of Calgary [Wyvill 86a] or with the Alias Modeller [Ali87], which models objects as spline surfaces, then triangulates them prior to rendering.





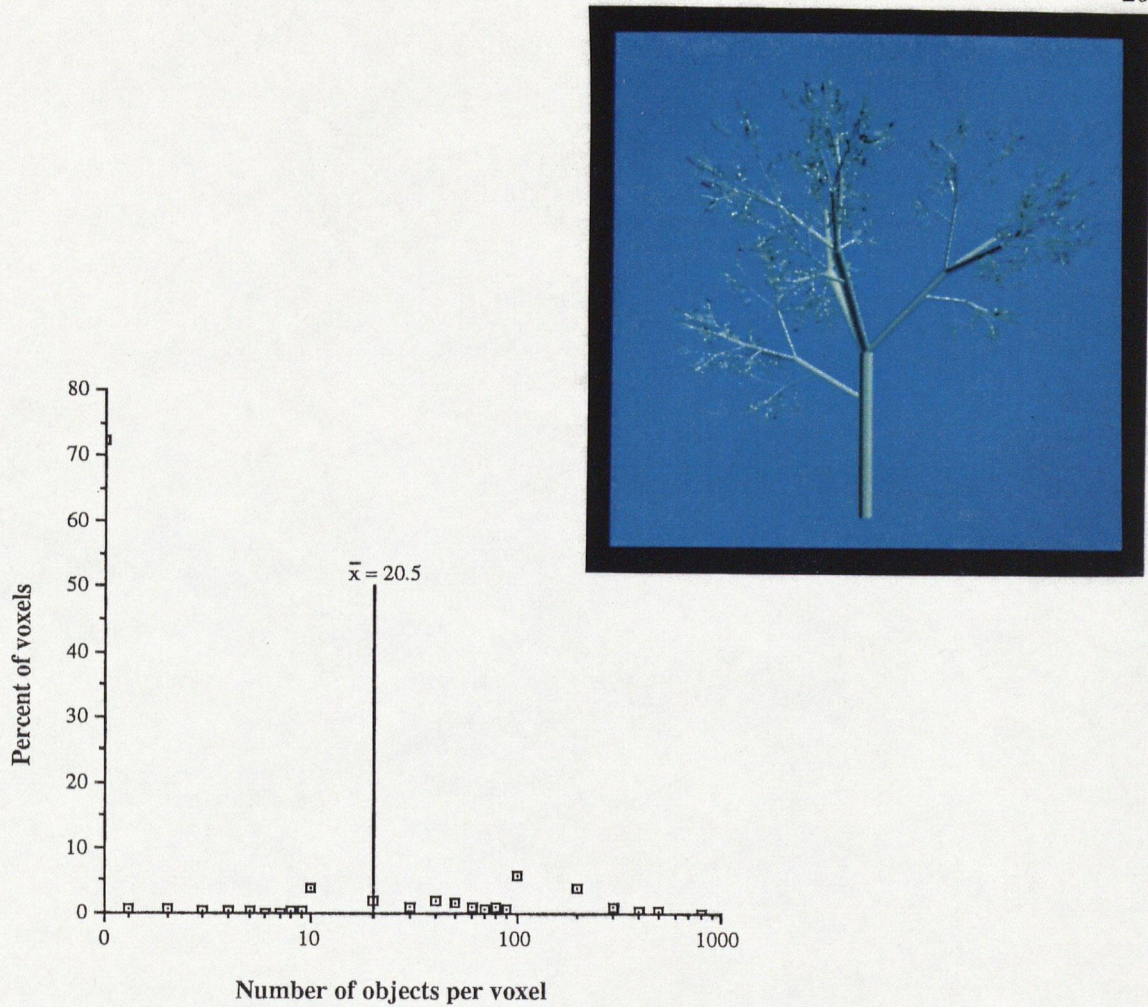
**Cubes.** 1,000 randomly oriented cubes, uniformly distributed on a  $10^3$  grid. The distribution of cubes is perfectly uniform, as is the area and perimeter of the polygons. This scene conforms to the assumptions made by Cleary and Wyvill in their analysis of uniform subdivision.

Number of objects = 6000, number of lights = 1, percent non-empty voxels = 100.

	$\bar{x}$	$\sigma$	$\sigma/\bar{x}$
no. of objects	6.0	0.0	0.0
object perimeter	0.4	0.0	0.0
object area	0.01	0.0	0.0

Figure 3.2: Cubes.





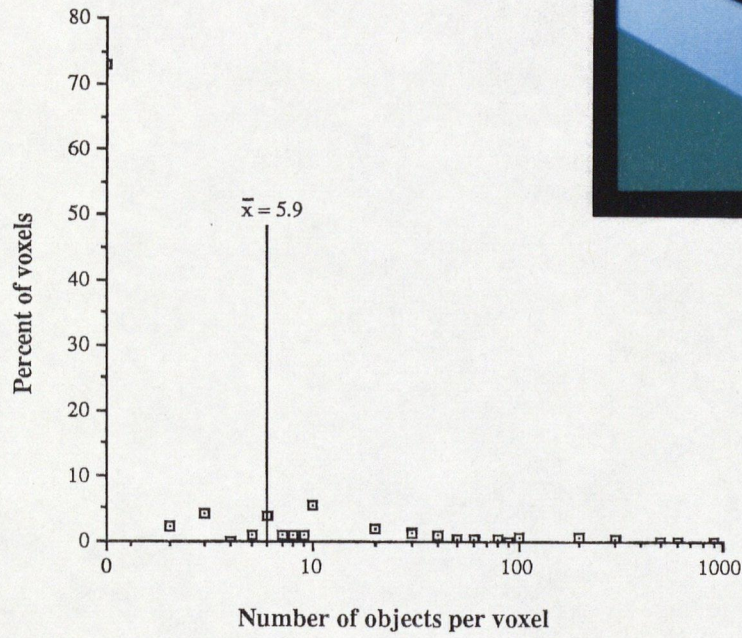
**Tree.** A recursively defined tree, similar to that used by Kay and Kajiya [Kay 86] and Scherson and Caspary [Scherson 87]. There are 1365 branches and 4096 leaves. Each branch is approximated by 12 polygons. This scene is much less uniform in object distribution than the cubes scene due to the empty spaces under and between the branches. It is not representative of complex scenes though, as the tree is in isolation with no background scenery.

Number of objects = 20476, number of lights = 1, percent non-empty voxels = 28.

	$\bar{x}$	$\sigma$	$\sigma/\bar{x}$
no. of objects	20.476	75.619	3.693
object perimeter	1.356	1.276	0.941
object area	0.032	0.114	3.547

Figure 3.3: Tree.





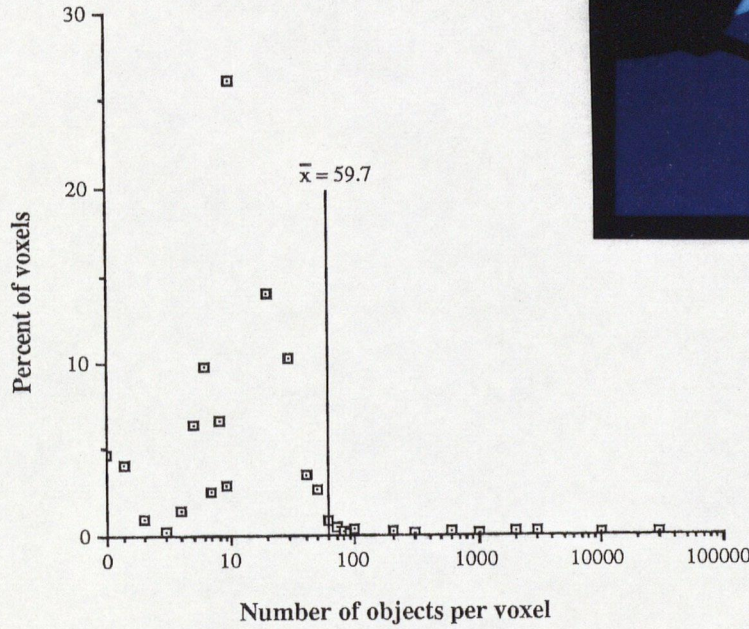
**Lumpy.** A frame from the short animated film “Lumpy’s Quest for Sev” [Jevans 89a]. This scene is typical of entertainment oriented animation. There is a small detailed model, the skateboarder, amid a much larger and less detailed back-drop.

Number of objects = 5938, number of lights = 6, percent non-empty voxels = 27.

	$\bar{x}$	$\sigma$	$\sigma/\bar{x}$
no. of objects	5.938	49.482	8.333
object perimeter	10.547	33.639	3.1289
object area	34.227	1040.278	30.393

Figure 3.4: Lumpy.





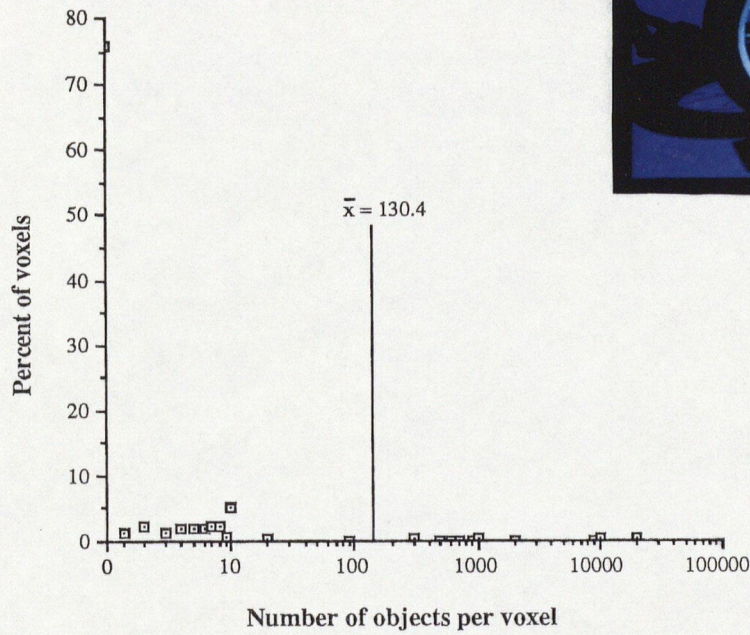
**Drum.** A much larger model than the first three. While 95.1 percent of the volume of the scene is non-empty, the standard deviation of the number of objects per voxel is large. This is because the drum, which occupies a small part of the scene, is composed of many small and complex polygons, while the room, which occupies most of the volume of the scene, is modelled by a few large polygons.

Number of objects = 59672, number of lights = 6, percent non-empty voxels = 95.1.

	$\bar{x}$	$\sigma$	$\sigma/\bar{x}$
no. of objects	59.672	1134.040	19.004
object perimeter	0.157	1.209	7.659
object area	0.010	0.291	28.902

Figure 3.5: Drum.





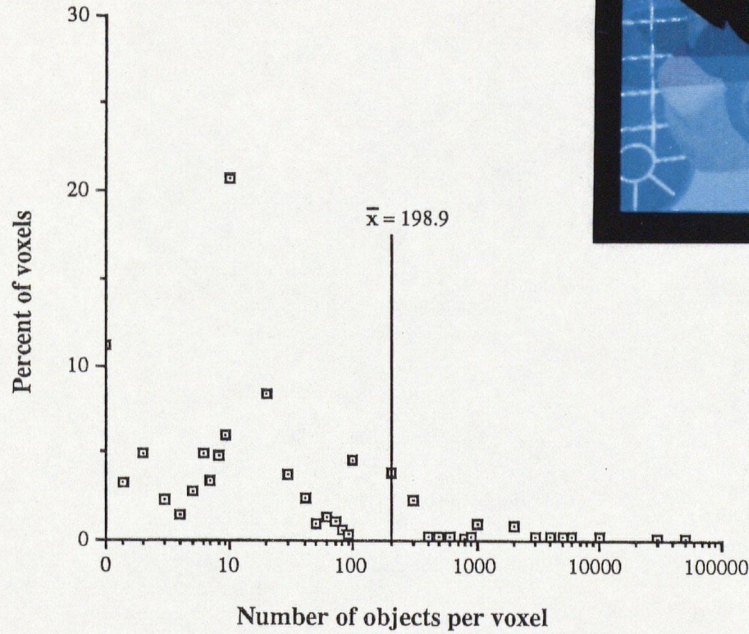
**Trike.** While the number of polygons is larger for the trike scene than for the drum scene, the standard deviation of the number of objects per voxel is lower. This is because the floor, modelled as several large polygons, does not occupy the large volume of space that the room in the drum scene does.

Number of objects = 130398, number of lights = 5, percent non-empty voxels = 23.6.

	$\bar{x}$	$\sigma$	$\sigma/\bar{x}$
<b>no. of objects</b>	130.398	1529.478	11.729
<b>object perimeter</b>	0.290	0.730	2.513
<b>object area</b>	0.022	0.442	19.705

Figure 3.6: Trike.





**Car.** This scene is taken from a commercial animation sequence. As with the drum scene, most of the voxels are non-empty, although the standard deviation of the number of objects per voxel is similar to that of the trike scene. This is because the windows and wall, which occupy a large volume of space, are more complex models than the simple walls in the drum scene.

Number of objects = 198941, number of lights = 7, percent non-empty voxels = 88.6.

	$\bar{x}$	$\sigma$	$\sigma/\bar{x}$
no. of objects	198.941	2255.443	11.337
object perimeter	0.929	3.483	3.746
object area	0.051	4.639	90.647

Figure 3.7: Car.



### 3.2.3 Observations

As evidenced by the normalised standard deviations measured in section 3.2.2, object distribution, area, and perimeter can vary greatly, even with scenes consisting of only a few thousand polygons.

The object distribution graphs show that scenes with similar standard deviations of object distribution can have very different scene structures. The **Trike** scene of figure 3.6 and the **Car** scene of figure 3.7 have normalised standard deviations of object distribution of approximately 11. Both the **Trike** and **Car** scenes exhibit areas of local uniformity of object distribution, combined with several very dense areas. The **Car** scene, however, has a large number of uniformly dense voxels, compared to a large number of empty voxels in the **Trike** scene.

These observations stimulate the need for an algorithm that makes use of uniform subdivision techniques in areas of local uniformity, yet can adapt to areas of high object density. The remainder of this thesis describes such an algorithm, and compares its performance with that of the octree and uniform subdivision algorithms.

## 3.3 Adaptive Voxel Subdivision

Octree subdivision can be generalised to hierarchical 3D grid subdivision by varying the subdivision granularity at each node according to the number and size of the objects in the corresponding voxel (figure 3.8). Snyder and Barr [Snyder 87] used this technique as a modelling tool as well as to accelerate ray tracing of tessellated surfaces. In their ray tracer, grids could be instantiated and referenced throughout a scene, allowing them to model a field containing billions of blades of grass from



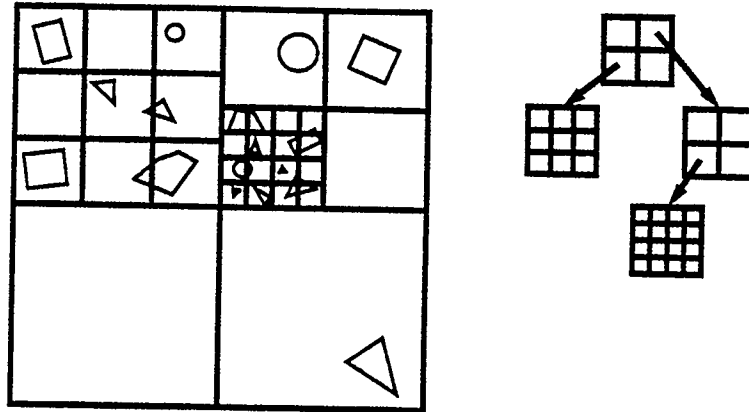


Figure 3.8: Hierarchical three dimensional grids.

just two primitives. Their method is not a general acceleration technique however, as it requires the grid hierarchies to be built by hand during the modelling stage. A new algorithm to accelerate ray tracing of arbitrary scenes, entitled *adaptive voxel subdivision*, is described below.

The adaptive voxel subdivision algorithm automatically constructs hierarchies of voxel grids according to the number of objects in each voxel. Rays are traced through the grids using a uniform grid traversal algorithm. A preliminary implementation of this algorithm was presented in [Jevans 89c]. The Cleary algorithm [Cleary 88] for traversal of a uniform grid was used, and details of vertical traversal, which are missing from the Fujimoto paper [Fujimoto 86], were given. Due to a deficiency in the implementation, the subdivision granularity inside each voxel was the same for the entire tree, and was set as a parameter of the program. A comparison of octree, uniform, and adaptive voxel subdivision schemes was made, although the results were not conclusive as to the superiority of the adaptive voxel subdivision algorithm over the other methods.

The following chapter describes a tool for implementing octree subdivision, uni-

form subdivision, and the new adaptive voxel subdivision. This updated implementation of the adaptive voxel subdivision algorithm uses heuristics, presented in section 5.4, to vary the subdivision inside each voxel according to the number of objects in it. An improved traversal algorithm and more efficient data structures are also detailed. Chapter 5 compares the performance of the octree, uniform, and adaptive voxel subdivision algorithms on the scenes that were analysed in section 3.2.

## Chapter 4

### A Space Subdivision Tool

This chapter describes an implementation of a ray tracer that uses hierarchical voxel grid subdivision to reduce the number of ray-object intersection tests required to ray trace an image. By imposing restrictions on the depth of the subdivision hierarchy and the subdivision granularity at each level, the ray tracer can implement octree, uniform, and the new adaptive voxel subdivision schemes.

Octree subdivision is achieved by restricting the granularity of subdivision at each level to  $2^3$ , and allowing the depth of the subdivision hierarchy to vary according to the number of objects in each voxel. Uniform subdivision is enforced by allowing the subdivision granularity of the root voxel to vary depending upon the number of objects in the scene, and by limiting the depth of the subdivision hierarchy to one. The adaptive voxel subdivision scheme presented in section 3.3 is enabled by allowing both the subdivision granularity and depth of the subdivision hierarchy to vary according to the number of objects in each voxel.

#### 4.1 Details of the Ray Tracer

The ray tracer, written in C++, implements the extended illumination model described in section 1.2.1. Reflection, refraction, and shadowing are supported by recursively applying the ray tracing routine to trace the secondary rays. To accelerate the tracing of shadow rays, a *shadow object cache* [Haines 86], described in

section 1.4.3, is stored at each light source. The maximum depth of the shade tree is limited to 4, usually producing good results for scenes with reflective or transparent objects.

#### 4.1.1 Anti-Aliasing

An adaptive anti-aliasing scheme is used to detect visible edges and to fire more rays per pixel along these edges [Mitchell 87]. A pixel that requires anti-aliasing is found by comparing its color value to those of the surrounding pixels. If the pixel's color differs significantly from those of its neighbors, it is super-sampled by firing more rays through it. These extra samples are then filtered with a box filter to reconstruct an anti-aliased color value for the pixel.

#### 4.1.2 Renderable Objects

The object-oriented implementation of the ray tracer allows the addition of new renderable object types without requiring recompilation of the entire program. Renderable objects must implement a method to calculate the bounding box for the object, used for insertion into voxels, a method to perform ray-object intersection tests, and a method that returns the surface attributes of an intersection point, for shading.

Two types of renderable objects, the polygon and the triangle, are implemented in the ray tracer. Ray-polygon intersection tests are performed using the ray-plane intersection and point-inside-polygon methods presented in [Foley 90](pages 339, 703-704). A subclass of the polygon, the triangle, uses the faster barycentric method outlined by Snyder and Barr [Snyder 87] to determine if a ray-plane intersection

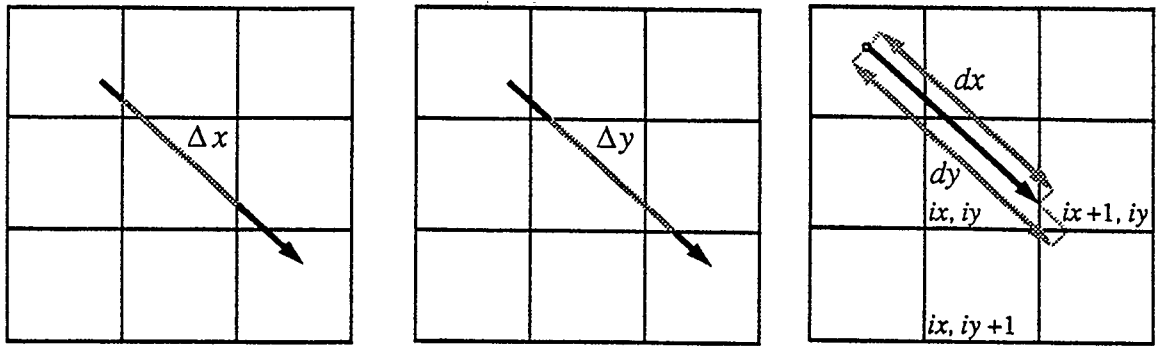


Figure 4.1: Cleary and Wyvill algorithm.

point lies inside a triangle. The ray-object intersection methods make use of ray signatures to avoid duplicating intersection tests.

## 4.2 Implementation of Ray Traversal

Each voxel may be subdivided by a uniform grid of sub-voxels, depending on the number of objects inside it. To traverse a ray through these grids, the uniform grid traversal algorithm of Cleary and Wyvill [Cleary 88] is used.

### 4.2.1 Horizontal Traversal

Consider figure 4.1, a two dimensional example. A ray in figure 4.1 enters a new voxel either by passing left to right through a vertical wall, or by passing from top to bottom through a horizontal wall. Considering only the passage through vertical walls, the distance along the ray between such crossings is a constant, labeled  $\Delta x$ . There is a similar constant distance,  $\Delta y$ , between successive crossings of horizontal walls, and, in the three dimensional case, a third constant,  $\Delta z$ .

To determine which voxel a ray moves to next, the wall that is crossed first must

```

initialize  $px, py, ix, iy, dx, dy, \Delta x$ , and  $\Delta y$ ;

while no intersection in voxel( $ix, iy$ )
  if  $dx \leq dy$  then
    begin
       $ix = ix + px$ ;
       $dx = dx + \Delta x$ ;
    end
  else if  $dx > dy$  then
    begin
       $iy = iy + py$ ;
       $dy = dy + \Delta y$ ;
    end;
end;

```

Figure 4.2: Pseudocode for the Cleary and Wyvill algorithm.

be determined. This can be done by keeping two variables,  $dx$  and  $dy$ , that record the total distance along the ray from some arbitrary origin to the next crossing of a vertical or horizontal wall. Note that  $dx$  and  $dy$  will always be positive. If  $dx$  is smaller than  $dy$ , then the next wall to be crossed is a vertical one, and the next voxel will be a horizontal neighbor. Correspondingly, if  $dy$  is smaller than  $dx$ , the next wall to be crossed is a horizontal one, and the next voxel will be a vertical neighbor.

Traversal variables  $dx$  and  $dy$  are maintained by incrementing  $dx$  by  $\Delta x$  whenever a vertical wall is crossed, and similarly incrementing  $dy$  by  $\Delta y$  whenever a horizontal wall is crossed. Every time the ray moves to a new voxel, the index of that voxel must be found. Let the index of a voxel be represented by variables  $ix$  and  $iy$ . When the ray moves to a new voxel, simply increment  $ix$  or  $iy$  by  $px$  or  $py$ , which are  $\pm 1$ , depending on the direction of the ray's travel. Pseudocode for the algorithm is given in figure 4.2.

### Algorithm Refinement

At each step in the traversal, the current voxel must be checked to see if it is empty or if it contains a sub-grid or a list of objects. This requires a lookup in a three

```

initialize  $px, py, i, dx, dy, \Delta x$ , and  $\Delta y$ ;

while no intersection in voxel( $i$ )
  if  $dx \leq dy$  then
    begin
       $i = i + px$ ;
       $dx = dx + \Delta x$ ;
    end
  else if  $dx > dy$  then
    begin
       $i = i + py$ ;
       $dy = dy + \Delta y$ ;
    end;
end;

```

Figure 4.3: Pseudocode for the revised Cleary and Wyvill algorithm.

dimensional array of voxels. In a computer's memory, a three dimensional array is stored as a one dimensional vector, and every index into the three dimensional array is converted into a one dimensional index, requiring three multiplications and two additions. If  $n$  is the size of the voxel space along one axis, the one dimensional index,  $i$ , corresponding to the three dimensional index  $ix, iy, iz$ , is calculated as  $i = ix * n * n + iy * n + iz$ .

To avoid the overhead of multiplications in the inner traversal loop, the voxel space can be explicitly stored and indexed as a one dimensional array of size  $n^3$ . The index of a voxel in this array,  $i$ , is incrementally maintained just as  $ix, iy$ , and  $iz$  were, except that the increments must be factors of  $n$ .  $px$  is set to  $\pm n^2$ ,  $py$  is  $\pm n$  and  $pz$  is  $\pm 1$ . Pseudocode for the revised algorithm is given in figure 4.3.

If the new voxel into which the ray has moved is empty, the traversal process continues. If the new voxel contains objects and is not subdivided, the ray must be tested for intersection with the objects in the voxel. If the voxel is subdivided, the algorithm must descend into the voxel to traverse the ray through its sub-grid, accomplished by a recursive call to the horizontal traversal routine.

### 4.2.2 Horizontal Traversal Initialisation

It is assumed that the ray originates within the bounding volume of the scene. If it does not, it is intersected with the scene's bounding box, and this point is used as the ray's origin. Each ray has an origin,  $rx, ry, rz$ , and a direction vector,  $\delta x, \delta y, \delta z$ . A voxel grid of  $n^3$  voxels is subdivided equally along the x, y, and z axes at a granularity of  $n$ . Because the bounding box for the scene may not be a cube, the size of a voxel in a grid may differ along the x, y, and z axes, and is given by  $sx, sy, sz$ . The following discussion deals only with the initialisation of the traversal variables for the x axis, but is trivially extended for the y and z axes.

Traversal is done in voxel space coordinates, so the origin of the voxel grid that is to be traversed is subtracted from the ray's origin, and stored as  $x, y, z$ . This translates the ray's origin so that  $0 \leq x \leq sx * n$ . The initial index of the ray in the voxel grid is given by:

$$ix = \text{truncate}(x/sx) \quad (4.1)$$

$\Delta x$  (see figure 4.4) is calculated as:

$$\Delta x = sx/\delta x \quad (4.2)$$

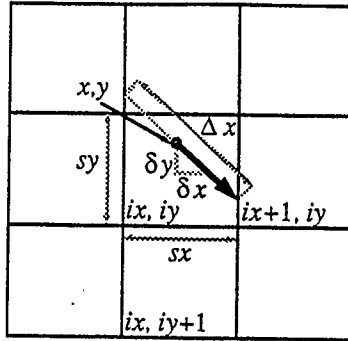
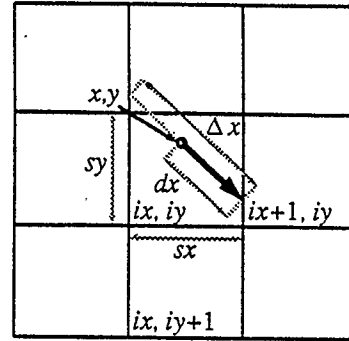
If the ray is moving in a positive direction along the x axis (figure 4.5) then  $dx$  is:

$$dx = ((ix + 1) * sx - x) * \Delta x \quad (4.3)$$

If the ray is traveling in a negative direction along the x axis then  $dx$  is:

$$dx = (x - ix * sx) * \Delta x \quad (4.4)$$



Figure 4.4: Initialising  $\Delta x$ .Figure 4.5: Initialising  $dx$ .

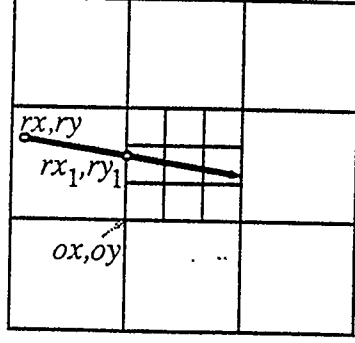
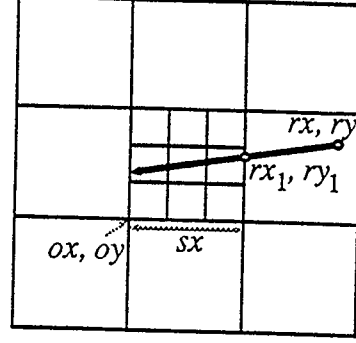
### 4.2.3 Vertical Traversal

During horizontal traversal, if a ray enters a voxel that is subdivided, the algorithm must descend into the subdivided voxel and traverse the ray through its sub-grid. This is accomplished by recursively calling the horizontal traversal routine.

#### Downwards Traversal

Before descending into a sub-grid, the intersection point of the ray with the subdivided voxel is found. This intersection point is used as the ray's origin,  $rx, ry, rz$  by the horizontal traversal initialisation code (section 4.2.2).

To quickly determine the intersection point of the ray with the voxel, the face of the voxel through which the ray entered must be found. This is done by keeping track, in the horizontal traversal loop, of the direction that the ray traveled to enter the current voxel. The axis of traversal is indicated by a variable,  $t$ , that is set at each iteration to 0, 1, or 2, corresponding to traversal in the  $x$ ,  $y$ , or  $z$  directions. The direction of travel is indicated by the sign of each element in the ray's direction

Figure 4.6: Finding  $rx_1, ry_1$ .Figure 4.7: Finding  $rx_1, ry_1$ .

vector. In figure 4.6,  $t = 0$  and  $\delta x$  is positive, indicating that the ray's movement was in the positive  $x$  direction.

Let  $ox, oy$  be the 2D origin, in world space coordinates, of the voxel that contains the sub-grid, and  $rx, ry$  be the origin of the ray. The intersection of the ray with the sub-voxel,  $rx_1, ry_1$ , is calculated as:

$$rx_1 = ox \quad (4.5)$$

$$ry_1 = ry + \frac{(ox - rx)}{\delta x} * \delta y \quad (4.6)$$

If  $\delta x$  is negative, indicating that the ray is traveling in a negative direction along the  $x$  axis, as in figure 4.7, then  $rx_1, ry_1$  are calculated as:

$$rx_1 = ox + sx \quad (4.7)$$

$$ry_1 = ry + \frac{(ox + sx - rx)}{\delta x} * \delta y \quad (4.8)$$

The intersection of the ray with the voxel,  $rx_1, ry_1$ , is used as its origin,  $rx, ry$ , during the horizontal traversal of the sub-grid.

These equations are easily extended to three dimensions. If  $t$  indicates that traversal was in the  $y$  or  $z$  directions, the equations are similar.

### Upwards Traversal

When a ray exits the bounds of a voxel grid, or when a ray-object intersection is found, horizontal traversal of the ray through the current voxel grid terminates. The status of the ray is indicated by the return value of the horizontal traversal routine. A return value of **0** indicates that the ray traveled beyond the bounds of the grid. A return value of **1** indicates that the ray intersected an object. If the traversal routine returns **0**, then horizontal traversal resumes in the parent voxel. If the horizontal traversal routine returns **0** from the root voxel, then the ray has left the scene without intersecting an object. If **1** is returned, all recursions terminate, and the intersection information is passed back to the shade routine.

#### 4.2.4 Lazy Subdivision

Rather than subdividing the scene in a preprocessing step before ray tracing, subdivision is done lazily, when a voxel is entered by a ray for the first time. This approach subdivides only areas of a scene through which rays pass.

Every voxel has a subdivision flag, initialised to **0** when the voxel is created, that indicates whether or not it has been considered for subdivision. When a voxel is entered by a ray, its subdivision flag is checked to see if it has been considered for subdivision. If the flag is **0**, then subdivision heuristics are consulted, possibly resulting in the voxel being subdivided. The flag is then set to **1**, and subsequent rays will not cause the voxel to be reconsidered for subdivision.

#### 4.2.5 Grid Data Structure

Because a voxel grid may be sparse, considerable memory savings can be had by storing only non-empty voxels. Both Pearce [Pearce 87] and Cleary and Wyvill [Cleary 88] stored non-empty voxels in a hash table, and used a voxel's one dimensional index as its key. Because an adaptive algorithm will have fewer empty voxels than a uniform subdivision algorithm, and since modern machines have large memory capacities, with 32 Mb not uncommon, the hash table scheme is abandoned in this implementation.

A voxel grid is stored as a one dimensional array of pointers to voxel structures, reducing both the complexity of the implementation and the overhead of checking for non-empty voxels during ray traversal. To check if a voxel is non-empty, the voxel grid array is indexed with the voxel's one dimensional index. If the voxel's entry in the array is non-null, then it is non-empty.

A voxel structure contains a subdivision flag to indicate whether or not the voxel has been considered for subdivision, and a second flag that indicates whether the voxel is a leaf node or is further subdivided. If the voxel is a leaf node, it contains an integer that indicates the number of objects inside it, and an array of pointers to those objects. If the voxel is subdivided, it contains an integer that indicates its subdivision granularity,  $g$ , and an array of size  $g^3$  of pointers to its sub-voxels.

## Chapter 5

### Algorithm Comparison

In this chapter, the implementation described in chapter 4 is used to measure the performance of the octree, uniform, and adaptive voxel subdivision algorithms. For each algorithm, the parameter that controls subdivision is varied, and its effect on the time to render an image is graphed. The maximum depth of the subdivision hierarchy is varied for the octree and adaptive voxel subdivision algorithms. Subdivision granularity is varied for the uniform voxel subdivision algorithm.

The performance of each algorithm is graphed for the six scenes presented in section 3.2. From these performance graphs, the subdivision parameter values that result in the fastest rendering time for each image are found. Section 5.5 compares the fastest rendering time of the algorithms for each image.

#### 5.1 Graphing Rendering Performance

A graph of rendering performance plots the subdivision parameter being varied along the horizontal axis, and the *scaled rendering time* of the image along the vertical axis. Scaled rendering time is the computation time (CPU time) to render an image divided by the CPU time of the irreducible ray tracing overhead, and is the ratio of the actual rendering time to the minimum possible rendering time.

The irreducible overhead of ray tracing an image is the time required to initialise each ray, and perform a single ray-object intersection calculation and shading calcu-

lation for each ray that intersects an object. This is the minimum possible rendering time for the ray tracing method, barring the use of ray coherence techniques. Scaling the rendering time by the irreducible overhead time gives a measure of how closely an algorithm approaches the minimum possible rendering time. A scaled ray tracing time of 1 indicates that an algorithm performs no computation to find the closest object intersected by each ray. The scaled rendering time is also useful for comparing the performance of algorithms run on different machines, because the speed of individual machines is factored out of the rendering time.

To calculate the irreducible overhead time of rendering an image, the image is ray traced in a pre-processing step, and a pointer to the object intersected by each ray is written to an *intersection file*. If a ray leaves the scene without intersecting an object, or if a shadow ray hits a light source without intersecting a shadowing object, then a null pointer is written.

The image is then rerendered using the data in the intersection file to give *a priori* knowledge of the object intersected by each ray. Rather than tracing a ray through the scene to determine the closest ray-object intersection, the object pointer for the ray is read from the intersection file. If the pointer is null, no ray-object intersection calculation is performed. If the pointer refers to an object, a ray-object intersection calculation is performed against that object. The time taken to render the image in this way is the irreducible ray tracing overhead time.

### 5.1.1 Rendering Details

Each image was rendered at 200 by 200 pixel resolution with one ray per pixel. All images were rendered on a SUN<sup>1</sup> 4/280 with 32 megabytes of RAM and 256 megabytes of swap space. User CPU time was measured, ensuring that I/O or swapping time was not included in the rendering time.

In each of the following sub-sections, the subdivision parameters of each algorithm are discussed; the six test images of section 3.2 are rendered with varying values of the subdivision parameters; and the results of this experimentation are analysed.

## 5.2 Octree Subdivision

Octree subdivision is effected by constraining the subdivision granularity of each voxel to be  $2^3$ , and by varying the maximum depth of the subdivision hierarchy. A voxel that is not at the maximum depth of the hierarchy is subdivided if the number of objects inside it is greater than a pre-defined *subdivision threshold*. This threshold must be chosen carefully, as it is a major factor influencing the subdivision of a scene.

### 5.2.1 Choosing the Subdivision Threshold

Scherson and Caspary [Scherson 87] attempted, unsuccessfully, to choose the subdivision threshold for their octree implementation by using theoretical analysis to determine the number of objects per voxel that results in the fastest rendering time.

---

<sup>1</sup>SUN is a trademark of Sun Microsystems, Inc.

To find the optimum number of objects in a voxel, the time to intersect a ray with the objects in an unsubdivided voxel is compared with the time to traverse a ray through a subdivided voxel. Let  $k$  be the ratio of the average number of traversed voxels to the number of voxels along a path parallel to a coordinate axis. Scherson and Caspary express the average time to process a ray as:

$$T = kg^{1/3}(n\alpha + OH) \quad (5.1)$$

where:

- $g$  = total number of voxels (granularity)
- $n$  = number of polygons in a voxel
- $\alpha$  = time to perform a ray-polygon intersection test
- $OH$  = processing time of a voxel (overhead)

If  $p$  is the total number of polygons in the scene, then  $p = gn$ . Substituting into equation 5.1 results in  $T$  being a function of a single variable,  $n$ . The optimal value of  $n$ ,  $n_{opt}$ , is found by differentiating  $T$  with respect to  $n$  and equating the derivative to zero:

$$n_{opt} = \frac{OH}{2\alpha} \quad (5.2)$$

For Scherson and Caspary's implementation, the ratio of voxel traversal overhead,  $OH$ , to ray-polygon intersection time,  $\alpha$ , was approximately 4 to 1, yielding  $n_{opt}$  of 2. The ray tracing implementation used in this thesis has an average ray-polygon intersection time of 27.5 microseconds. The time to traverse a ray through a subdivided voxel, determined by tracing a random sampling of rays through a  $2^3$  grid, is an average of 83.1 microseconds. Thus the ratio of voxel traversal overhead to



ray–polygon intersection time is approximately 3 to 1, meaning that the threshold for subdivision lies at slightly less than 2 polygons per voxel.

The validity of equation 5.1 is suspect, since it does not consider the effects of object fragmentation or object size. This suspicion is confirmed by Scherson and Caspary themselves, as they experimentally determined that the optimal number of polygons in a voxel was approximately 7. Because the ray tracing implementation used in this thesis has a ratio of voxel traversal overhead to ray–polygon intersection time that is similar to that of Scherson and Caspary’s implementation, 7 is chosen as the subdivision threshold for the octree implementation.

### **5.2.2 Octree Performance**

The six scenes analysed in section 3.2 are rendered using octree subdivision. For each image, the maximum depth of the subdivision hierarchy is varied and plotted against the scaled rendering time.

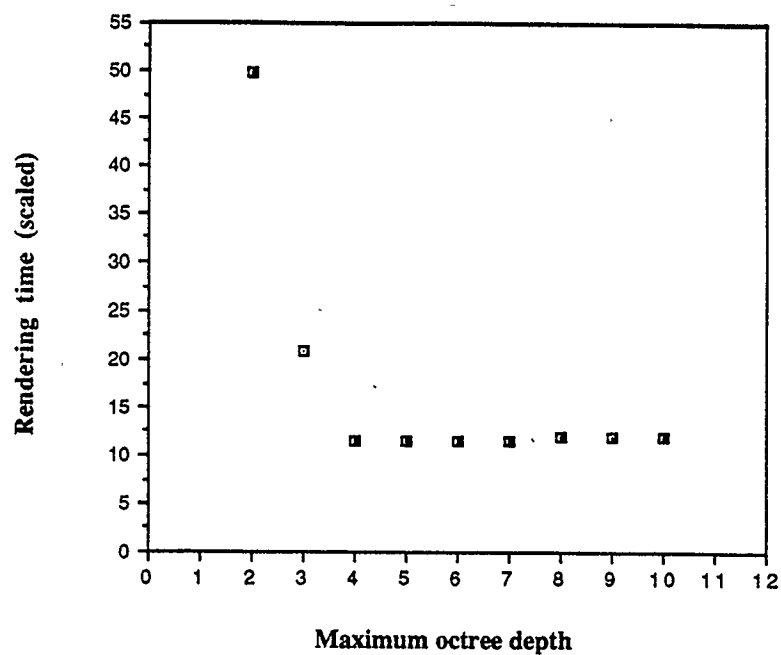


Figure 5.1: Octree performance on the **Cubes** scene.

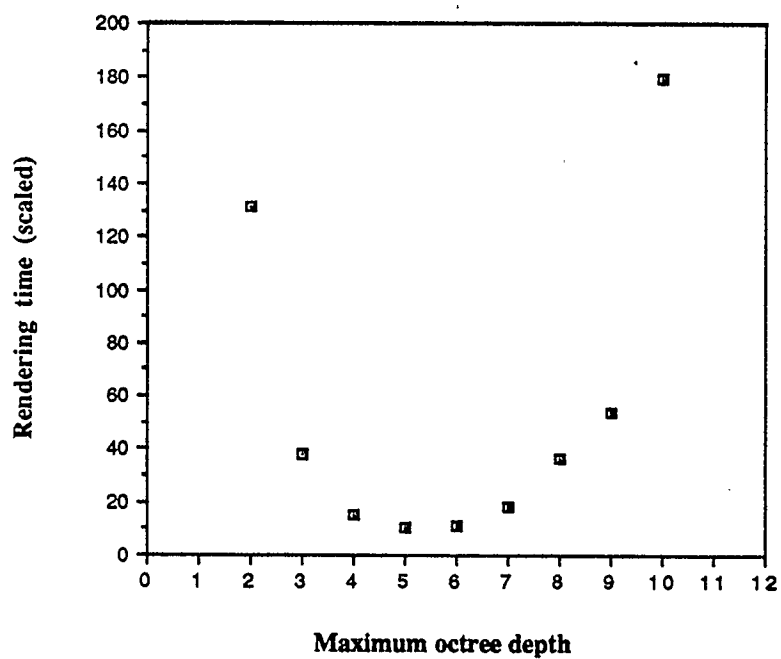


Figure 5.2: Octree performance on the **Tree** scene.

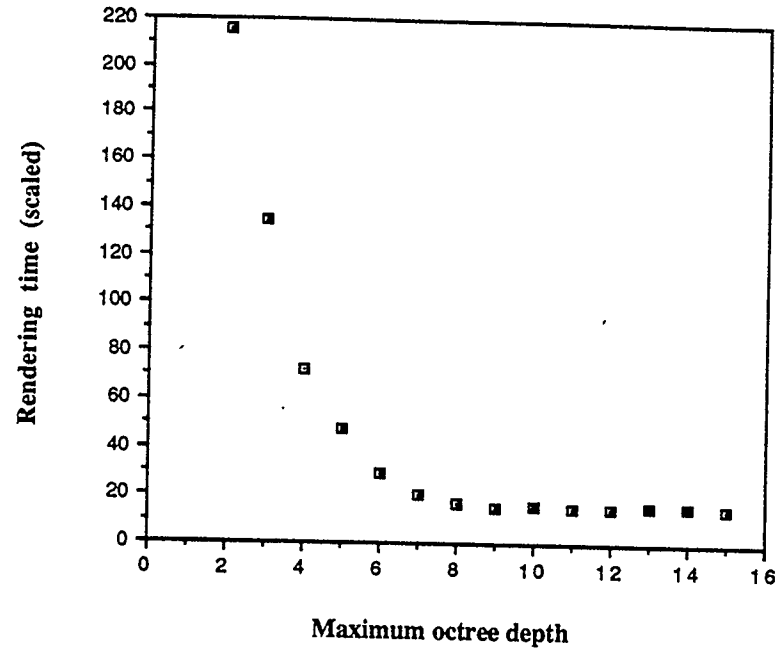


Figure 5.3: Octree performance on the **Lumpy** scene.

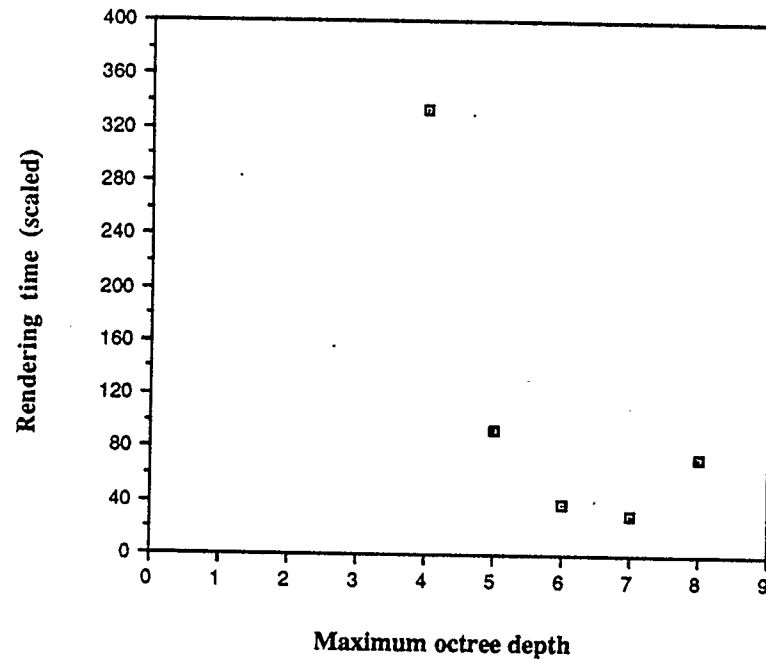


Figure 5.4: Octree performance on the **Drum** scene.

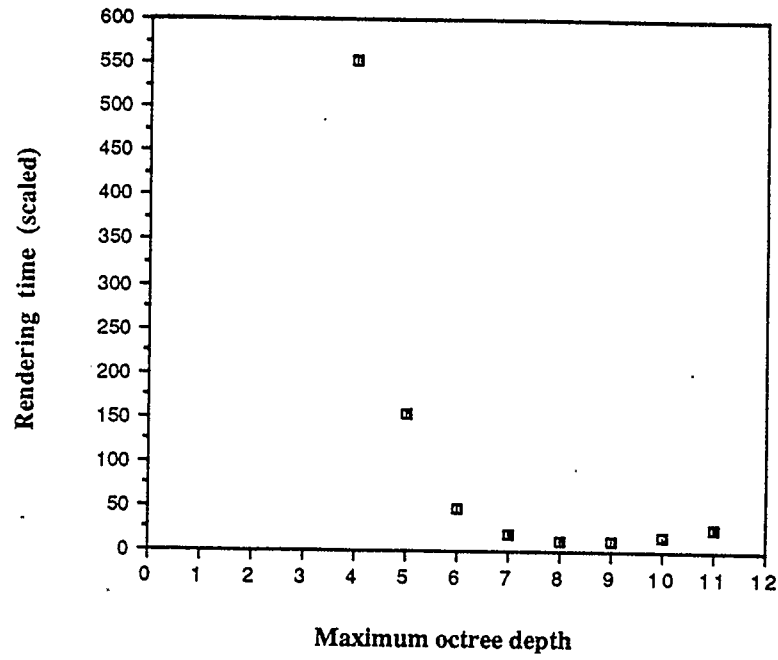


Figure 5.5: Octree performance on the **Trike** scene.

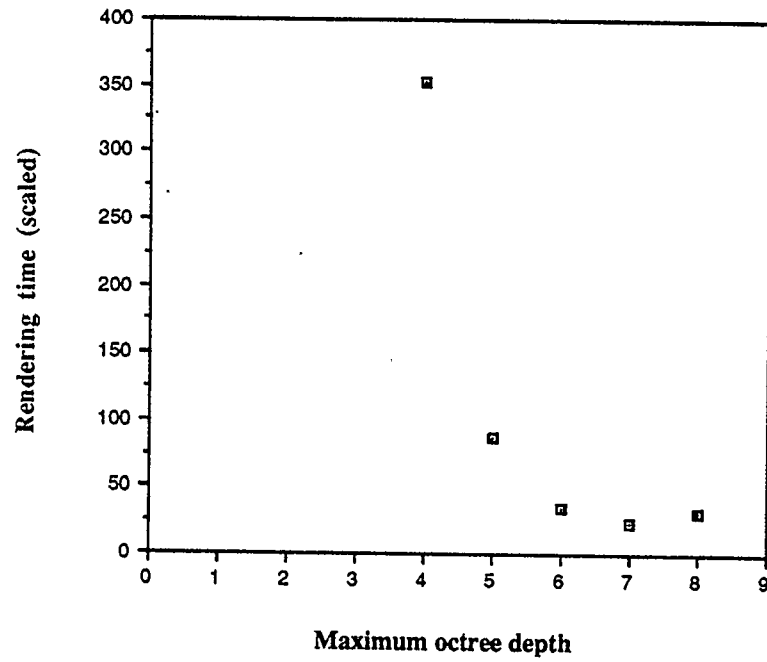


Figure 5.6: Octree performance on the **Car** scene.

As these graphs indicate, ray tracing time rapidly decreases as the maximum subdivision tree depth increases from zero. The time to render the **Cubes** and **Lumpy** scenes decreases to a minimum and remains there as the maximum tree depth parameter is increased. This is because, at some tree depth, there eventually becomes less than eight objects in every leaf voxel, and increasing the maximum subdivision hierarchy depth parameter has no effect on the subdivision.

The other four scenes exhibit a somewhat different performance characteristic. As maximum subdivision depth increases, the rendering time rapidly decreases to a minimum, and then begins to increase. This is caused by over-subdivision, which occurs when a voxel has more than the threshold number of objects, yet further subdivision does not significantly reduce the number of objects in the child voxels. Over-subdivision causes an increase in rendering time due to the increased time required to subdivide the voxels, and the increased horizontal and vertical traversal required to traverse a ray through an over-subdivided voxel.

There are two causes of over-subdivision. The first is when objects are large compared to the size of a voxel, and will lie in many of the voxel's children if it is

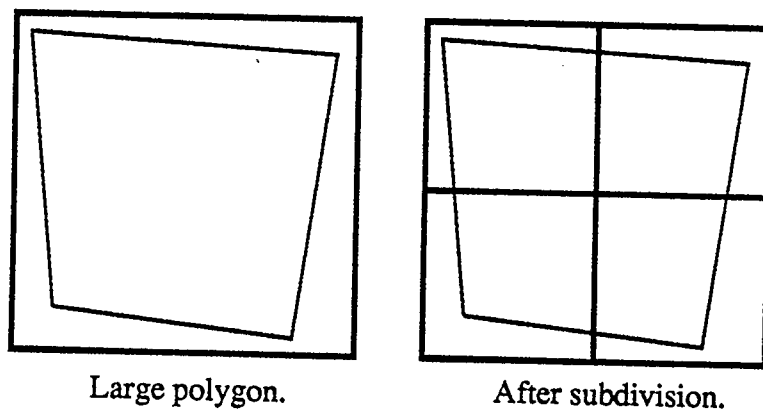


Figure 5.7: Over-subdivision of a large object.

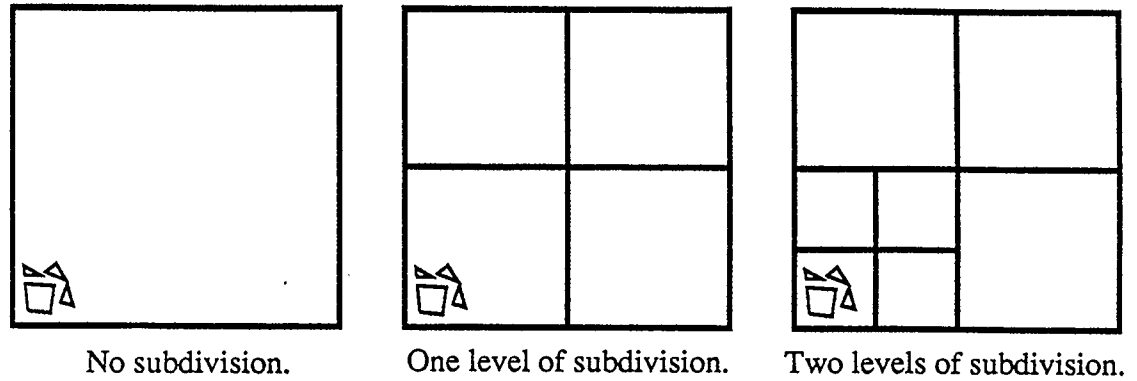


Figure 5.8: Over-subdivision of small objects.

subdivided (figure 5.7). The second cause of over-subdivision is when the objects in a voxel are very small compared to the size of the voxel, and a number of recursive subdivisions are required to reduce the number of objects in a leaf voxel (figure 5.8).

### 5.3 Uniform Subdivision

Uniform subdivision is achieved by restricting the depth of the subdivision hierarchy to one, thereby allowing only the root voxel to be subdivided. To measure the performance of uniform subdivision, the scenes analysed in section 3.2 are rendered a number of times with increasing subdivision granularities. The performance graphs plot scaled rendering time on the vertical axis, and the subdivision granularity along each of the x, y, and z axes of the scene, on the horizontal axis.

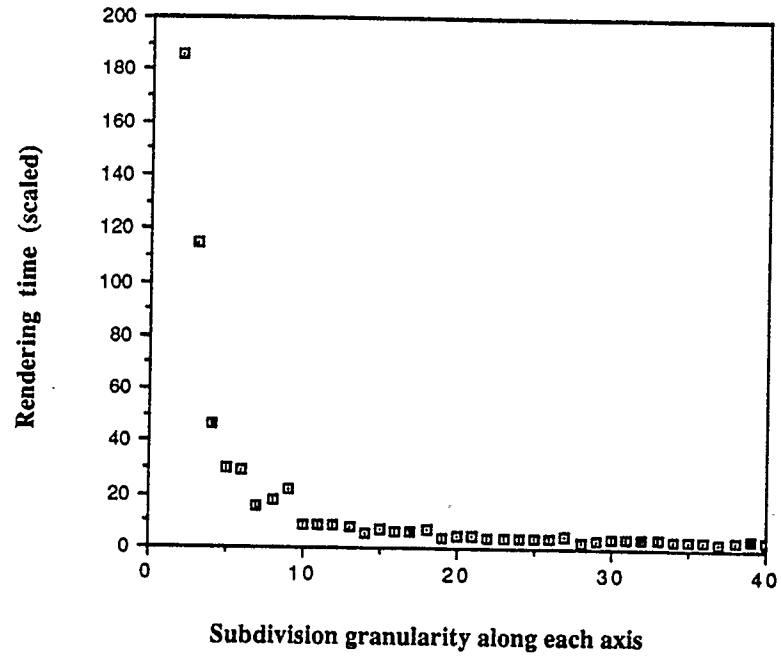


Figure 5.9: Uniform subdivision performance on the **Cubes** scene.

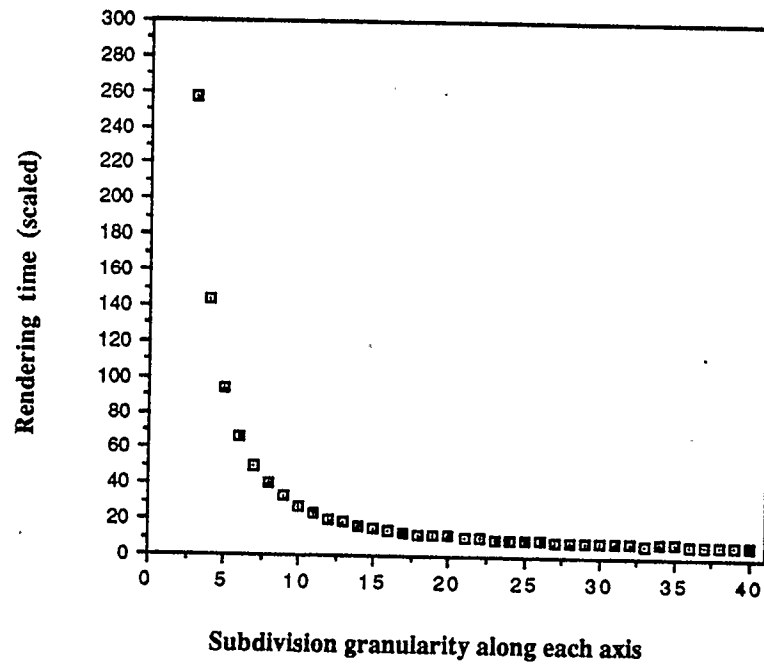


Figure 5.10: Uniform subdivision performance on the **Tree** scene.

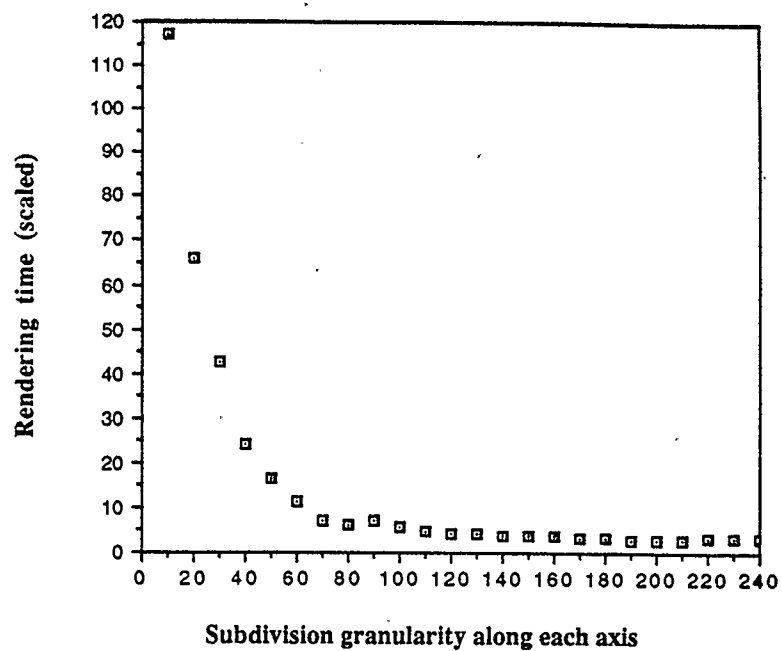


Figure 5.11: Uniform subdivision performance on the **Lumpy** scene.

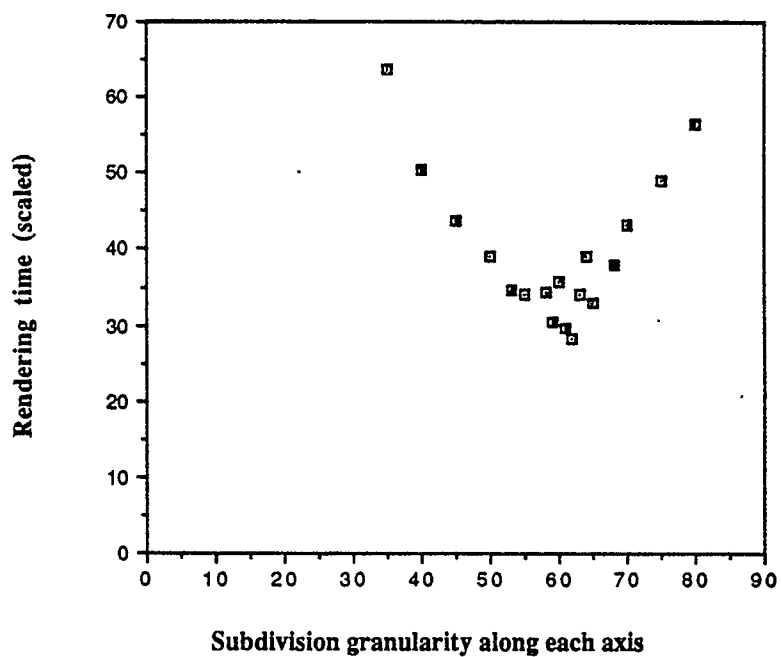


Figure 5.12: Uniform subdivision performance on the **Drum** scene.



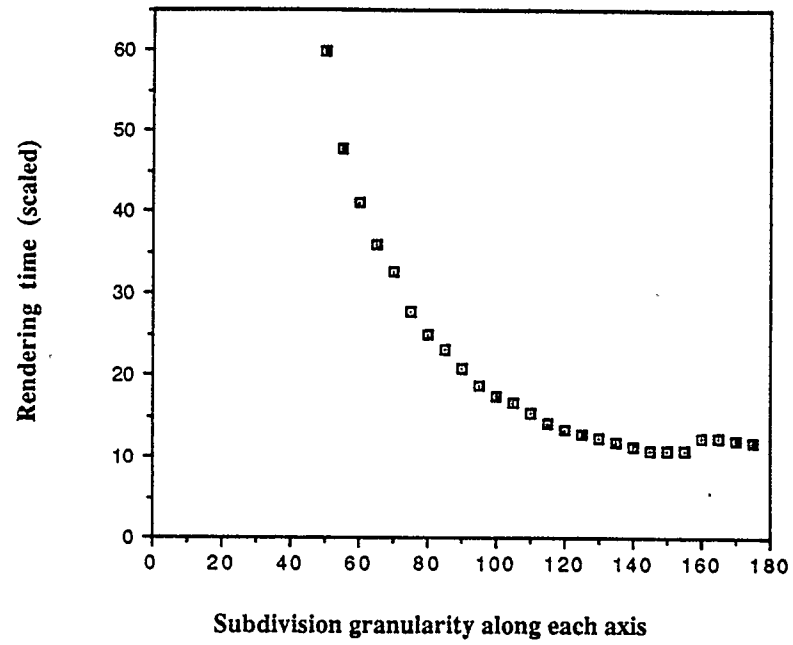


Figure 5.13: Uniform subdivision performance on the **Trike** scene.

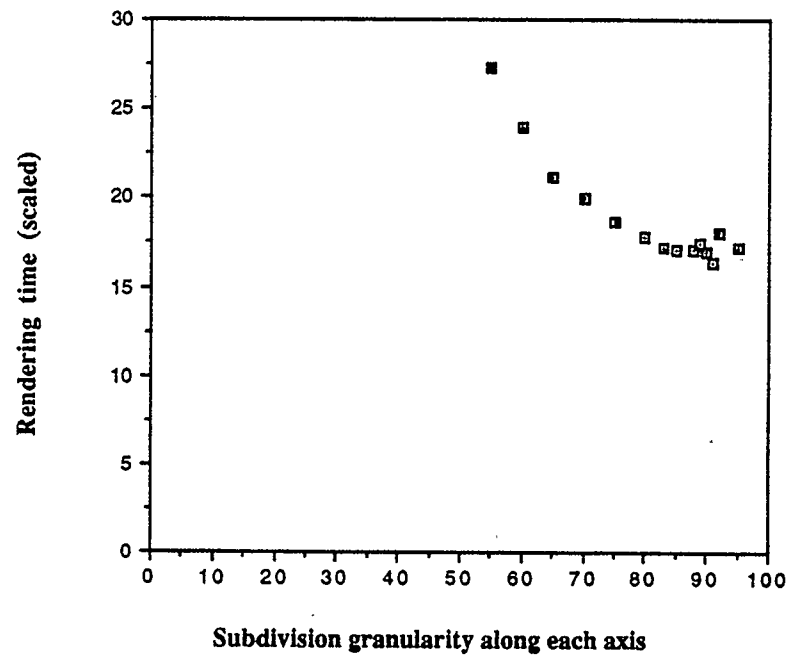


Figure 5.14: Uniform subdivision performance on the **Car** scene.

As with the octree method, rendering time decreases to a minimum as subdivision granularity increases. Unlike octree subdivision, the rendering time using uniform subdivision will always increase as the subdivision granularity exceeds the optimum, due to the increased costs of ray traversal, object fragmentation, and subdivision. This pattern is not as dramatic as with octree subdivision however, since the costs of fragmentation and ray traversal grow slower than for octrees. This is because the size of a leaf voxel in an octree decreases by a factor of eight with every increase in subdivision depth, whereas the decrease in voxel size is much more gradual as uniform subdivision granularity increases. Another factor in the more rapid increase in the costs of over-subdivision in an octree is that traversing a ray through an over-subdivided voxel entails vertical traversal, which is not required when traversing a uniform grid.

An interesting effect can be observed in the **Cubes** (figure 5.9), **Tree** (figure 5.10), and **Lumpy** (figure 5.11) graphs, where rendering time decreases in a step-like manner as subdivision granularity approaches the optimum. This is due to the effect of increased subdivision granularity on large polygons. As granularity increases, a large polygon will lie in more voxels, becoming more fragmented. As granularity increases further, the polygon will eventually be culled from some of the voxels, offsetting the effects of the increased fragmentation and ray traversal. An example is shown in figure 5.15.

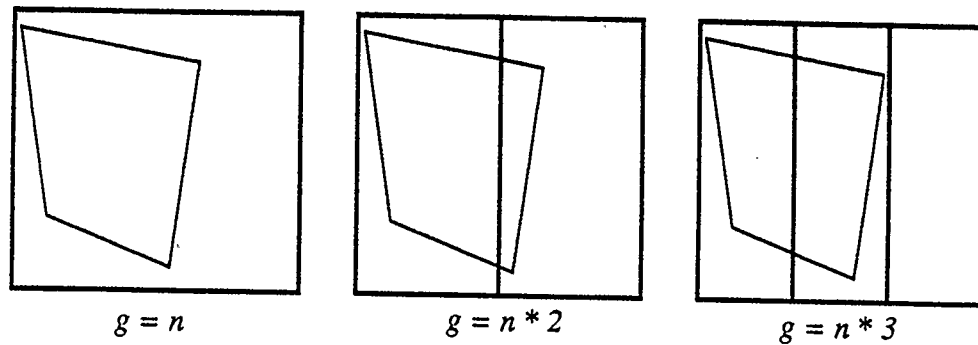


Figure 5.15: The effect of increasing subdivision granularity on a large polygon.

## 5.4 Adaptive Voxel Subdivision

Adaptive voxel subdivision is achieved by hierarchically subdividing a scene and allowing the subdivision granularity inside each voxel to vary depending on the number of objects inside it. As with the octree, a heuristic is used to control subdivision inside each voxel.

### 5.4.1 A Subdivision Heuristic

Cleary and Wyvill [Cleary 88] found that an effective uniform subdivision granularity for many scenes was  $N^{\frac{1}{3}}$  voxels on a side, where  $N$  is the number of objects in the scene. This motivates a subdivision heuristic that subdivides a voxel into  $n$  sub-voxels,  $n^{\frac{1}{3}}$  on a side, where  $n$  is the number of objects in the voxel. Analogous to the octree subdivision threshold of section 5.2.1, a voxel is not subdivided if it contains fewer than eight objects.

### 5.4.2 Performance of Adaptive Voxel Subdivision

The scenes analysed in section 3.2 are rendered a number of times with increasing maximum subdivision hierarchy depths.

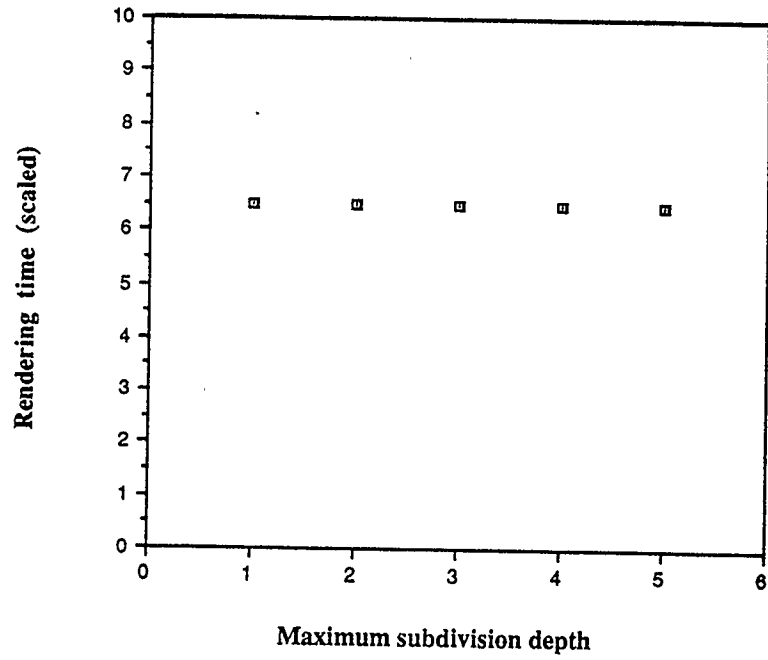


Figure 5.16: Adaptive voxel subdivision performance on the **Cubes** scene.

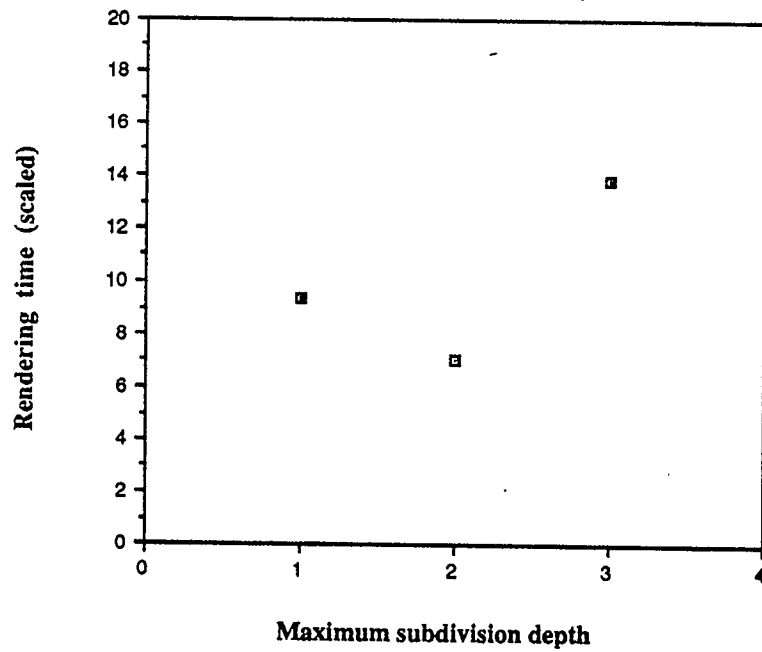


Figure 5.17: Adaptive voxel subdivision performance on the **Tree** scene.

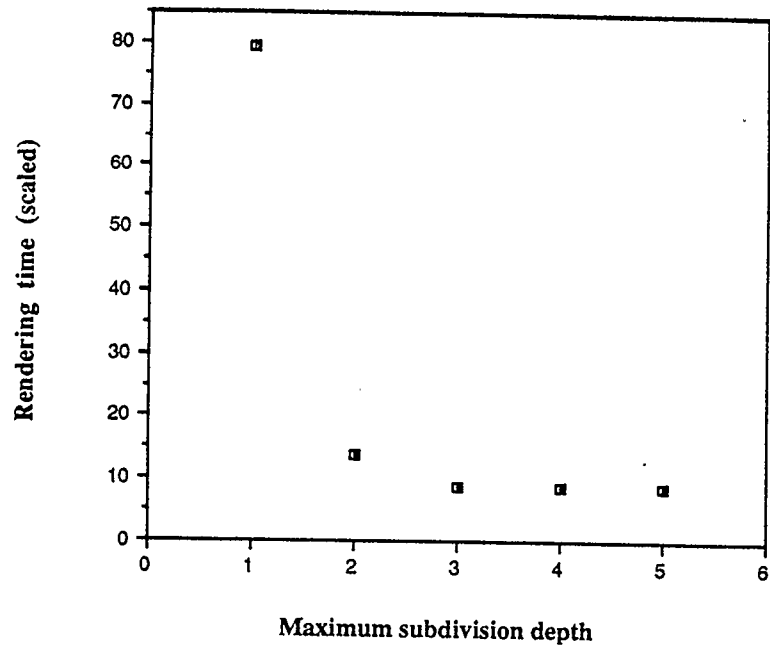


Figure 5.18: Adaptive voxel subdivision performance on the **Lumpy** scene.

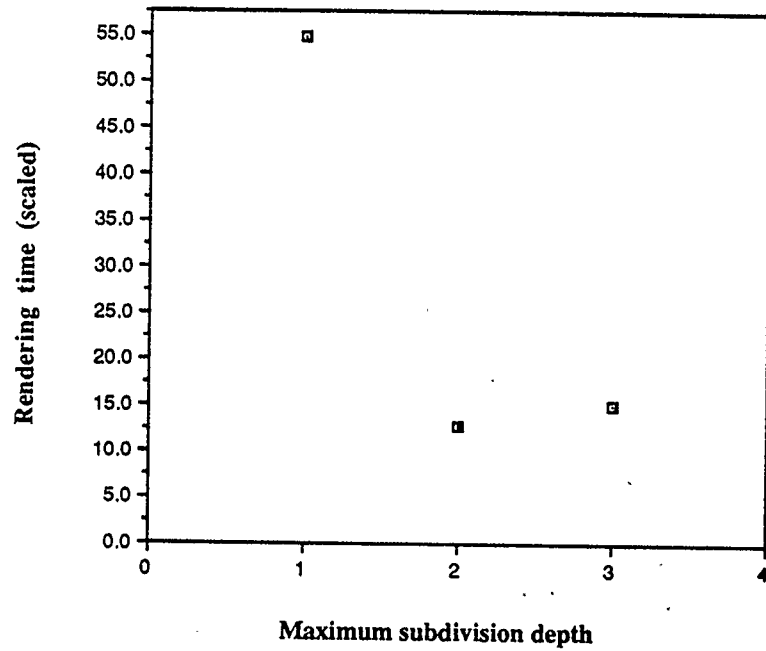


Figure 5.19: Adaptive voxel subdivision performance on the **Drum** scene.

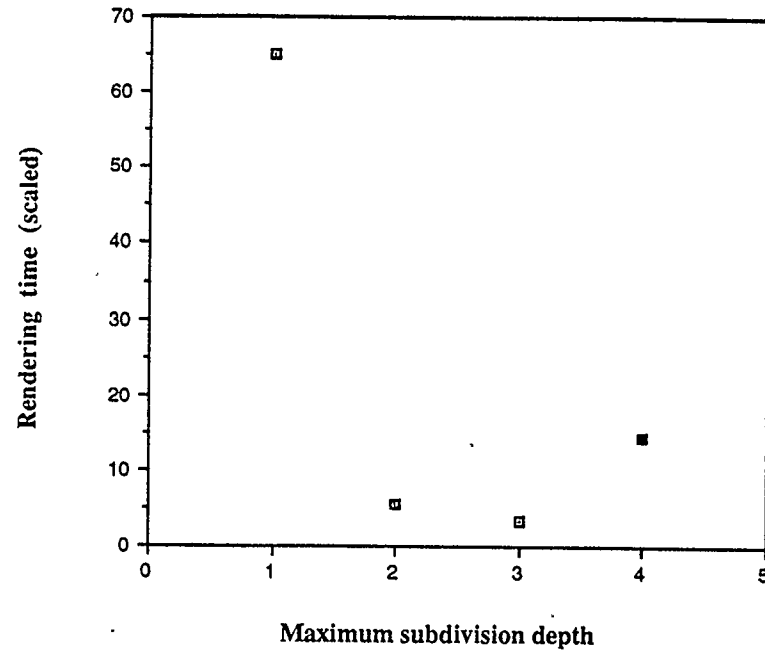


Figure 5.20: Adaptive voxel subdivision performance on the **Trike** scene.

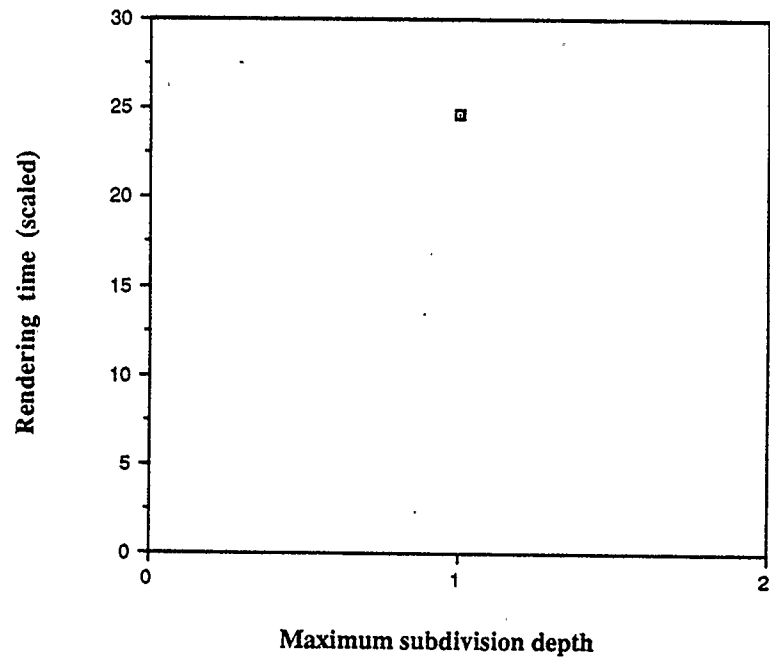


Figure 5.21: Adaptive voxel subdivision performance on the **Car** scene.

The performance of the adaptive voxel subdivision algorithm resembles that of the octree algorithm, but the minimum rendering time occurs at a lesser subdivision depth. As the maximum depth of the subdivision hierarchy increases beyond the optimum, rendering time increases much more quickly than with the octree algorithm, because the size of over-subdivided voxels can decrease by more than a factor of eight at each level.

The ray tracer runs out of memory when rendering the **Car** image at a subdivision depth greater than 1 (figure 5.21). The subdivision heuristic, operating under the assumption that the object distribution is uniform, subdivides the root voxel at a granularity of  $N^{\frac{1}{3}}$  voxels on a side, leaving no memory for further recursive subdivision.

#### 5.4.3 A Revised Heuristic

As seen in section 3.2.2, large scenes are unlikely to exhibit uniformity of object distribution. Also, as the number of objects in a scene increases, so does the likelihood of rendering a subset of the scene, because the camera is likely to be inside the scene, viewing a part of it. This is the case with the **Lumpy**, **Drum**, and **Car** images. These considerations motivate the development of a subdivision heuristic that is more adaptive than that of section 5.4.1.

The new subdivision heuristic subdivides a voxel at  $n^{\frac{1}{3}}$  granularity on a side when  $n$  is less than 1000. When  $n$  is between 1500 and 3500, the voxel is more finely subdivided, based on the observation that it is better to over-subdivide with a uniform grid, than to under-subdivide, since the cost of over-subdivision grows very slowly. A maximal subdivision granularity of  $20^3$  is reached when there are between

2000 and 3500 objects in a voxel. As  $n$  grows larger than 5000, the granularity of subdivision decreases, because the likelihood of a uniform distribution of objects is small. When there are more than 10000 objects in a voxel, the subdivision granularity remains fixed at  $5^3$ .

A table of these experimentally derived heuristics, compiled with the aid of Andrew Pearce, is presented below.

$n$	$g$	$n$	$g$
8	$2^3$	1500	$15^3$
27	$3^3$	2000	$20^3$
64	$4^3$	3500	$15^3$
100	$5^3$	5000	$10^3$
216	$6^3$	6000	$9^3$
343	$7^3$	7000	$8^3$
512	$8^3$	8000	$7^3$
729	$9^3$	9000	$6^3$
1000	$10^3$	10000	$5^3$

$n$  = the number of objects in a voxel

$g$  = the subdivision granularity of the voxel

### Results Using the Revised Heuristics

The scenes analysed in section 3.2 are rendered a number of times with increasing maximum subdivision hierarchy depths.



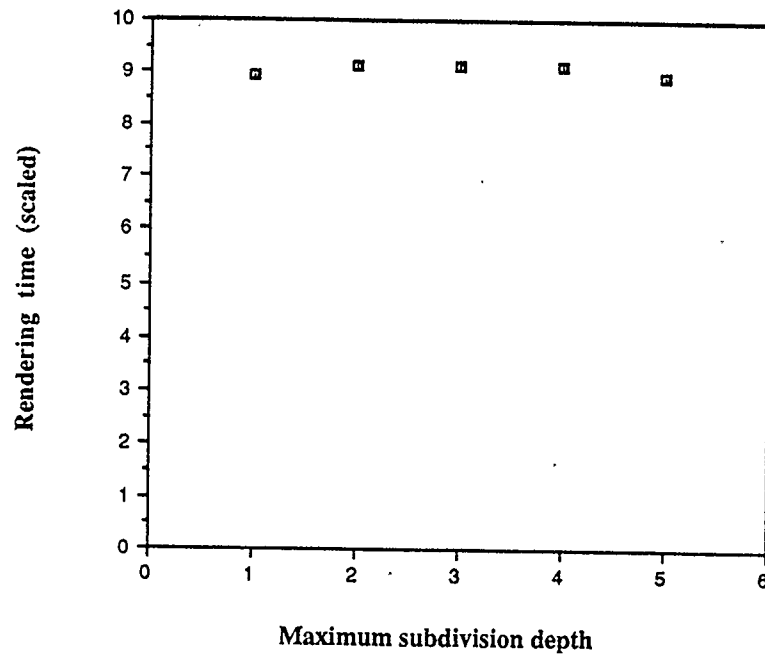


Figure 5.22: Revised adaptive voxel subdivision performance on the **Cubes** scene.

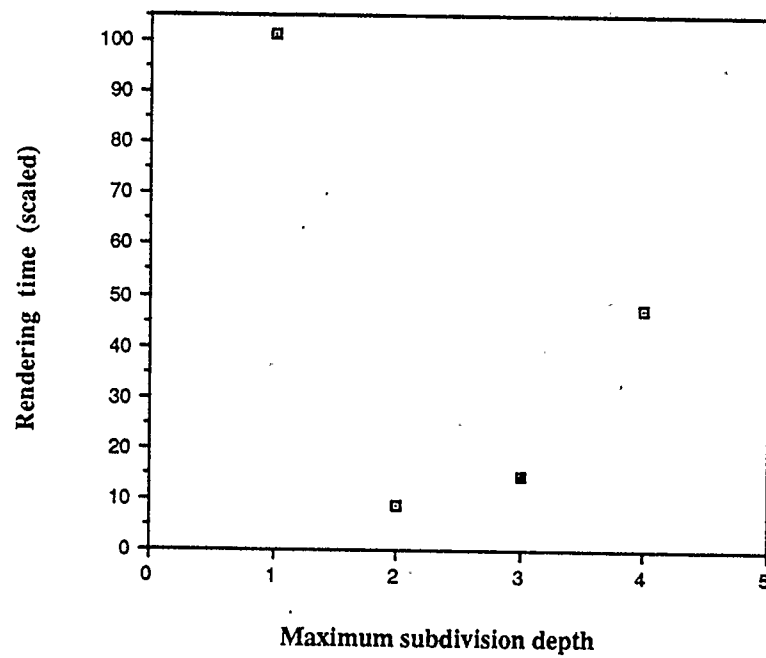


Figure 5.23: Revised adaptive voxel subdivision performance on the **Tree** scene.

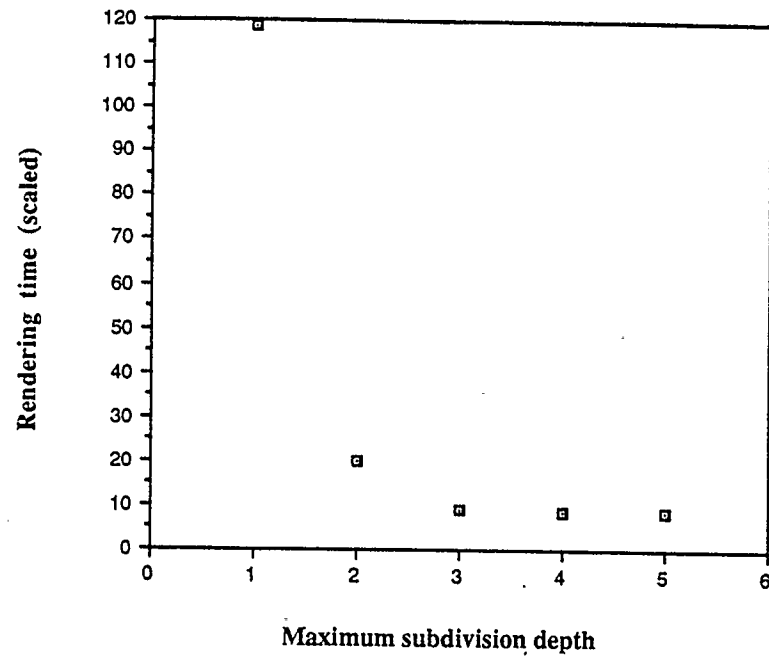


Figure 5.24: Revised adaptive voxel subdivision performance on the **Lumpy** scene.

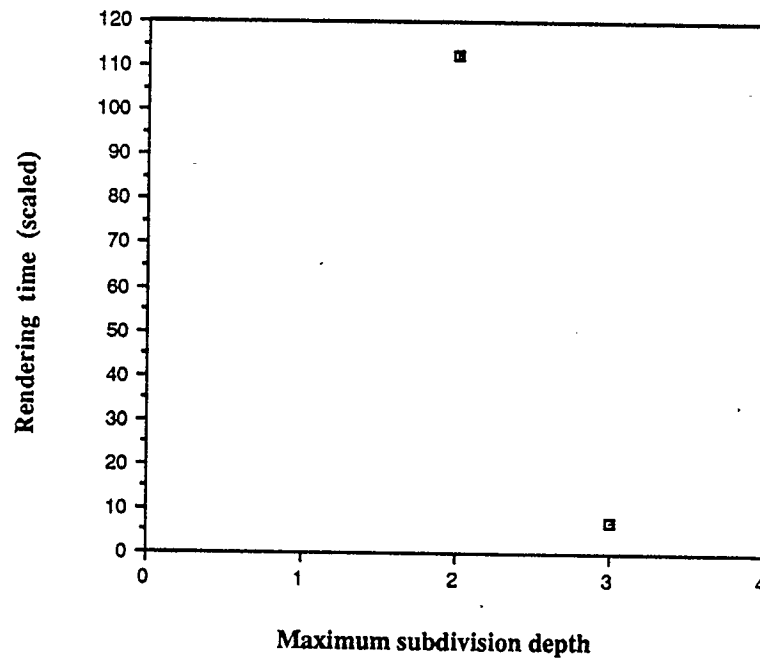


Figure 5.25: Revised adaptive voxel subdivision performance on the **Drum** scene.

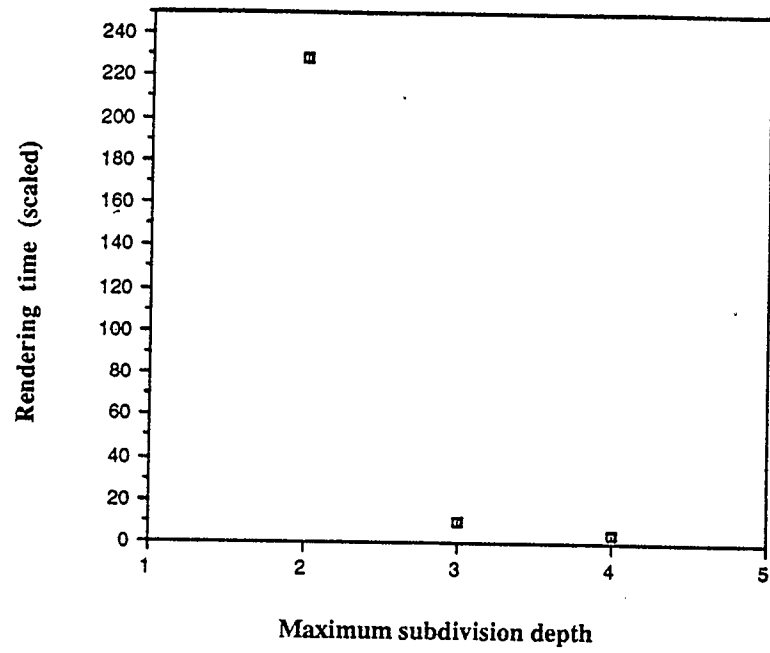


Figure 5.26: Revised adaptive voxel subdivision performance on the **Trike** scene.

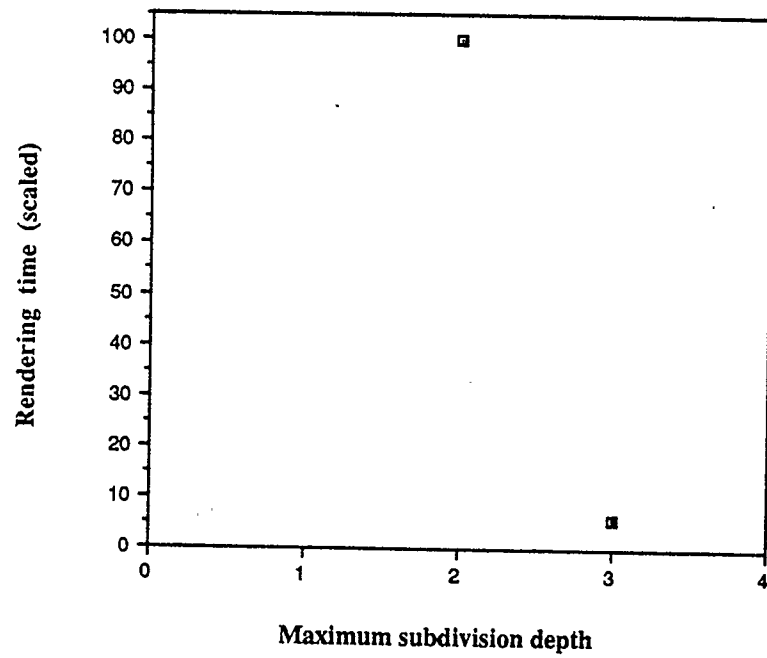


Figure 5.27: Revised adaptive voxel subdivision performance on the **Car** scene.

The results using the revised heuristic are similar to those obtained with the first heuristic, although the subdivision hierarchy depth that results in minimal rendering time is usually greater with the revised heuristic. The revised heuristic allows the **Car** image (figure 5.27) to be rendered at a subdivision depth of 3 without running out of memory. The **Drum**, **Trike**, and **Car** images cannot be rendered with a maximum subdivision depth larger than is optimal, as the ray tracer runs out of memory.

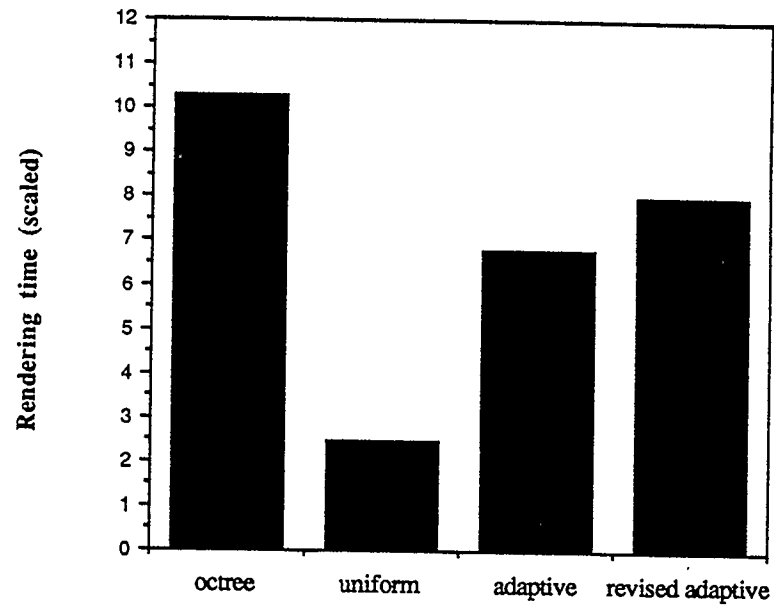
## 5.5 Comparing the Algorithms

To determine the relative effectiveness of the algorithms examined in this chapter, their fastest rendering performance for each of the six test images is compared. For each algorithm, the six images are rerendered at 512 by 512 pixel resolution with one ray per pixel, using the subdivision parameter value that gives the fastest rendering time. A bar graph comparing the scaled fastest rendering times of the four algorithms is plotted for each image.

The irreducible overhead time and the number of rays traced by the irreducible overhead method are given for each image. For each algorithm, the value of the optimal subdivision parameter, the rendering time, and the amount of memory required for the subdivision data structure are tabulated.

### 5.5.1 Comparison of Performance for the Cubes scene

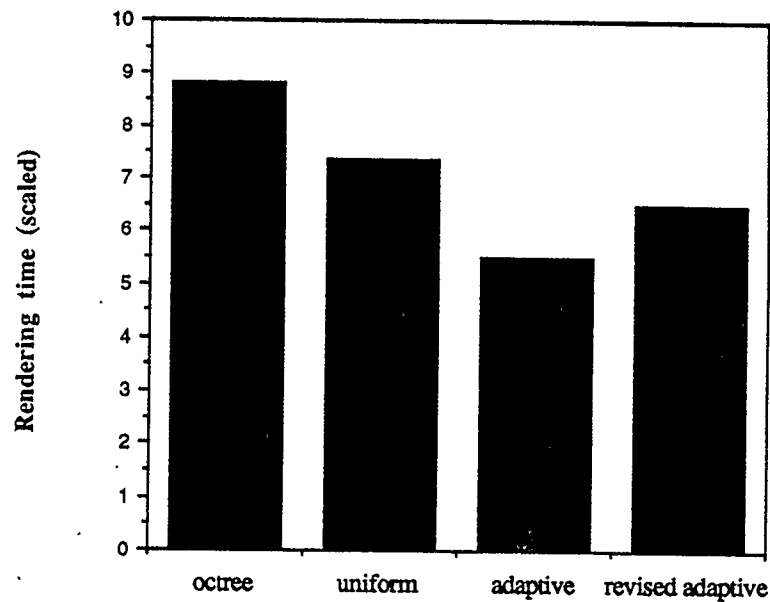
As expected for a scene with a uniform distribution of objects, the uniform subdivision algorithm has the fastest rendering time of the four algorithms. The adaptive



no. rays	irreducible overhead time
240393	138 sec.

algorithm	subdivision	cpu time	scaled time	memory
octree	depth = 4	1421 sec.	10.30	2.7Mb
uniform	$g = 28^3$	338 sec.	2.45	2.6Mb
adaptive	depth = 1	938 sec.	6.80	2.9Mb
revised adaptive	depth = 1	1103 sec.	7.99	2.5Mb

Figure 5.28: Comparison of algorithm performance for the **Cubes** scene.



no. rays	irreducible overhead time
254961	146 sec.

algorithm	subdivision	cpu time	scaled time	memory
octree	depth = 5	1289 sec.	8.83	11.0Mb
uniform	$g = 30^3$	1076 sec.	7.37	11.0Mb
adaptive	depth = 2	803 sec.	5.50	11.0Mb
revised adaptive	depth = 2	984 sec.	6.74	11.0Mb

Figure 5.29: Comparison of algorithm performance for the **Tree** scene.

voxel subdivision algorithm using the first heuristic (section 5.4.1), has the next best performance, since it subdivides the root voxel more finely than when using the revised heuristic. Predictably, the octree algorithm has the slowest rendering time, since the optimal tree depth is four, meaning that a significant amount of vertical traversal is required to trace a ray through the scene.

### 5.5.2 Comparison of Performance for the **Tree** scene

The **Tree** scene has a fairly uniform distribution of objects, indicated by its nor-

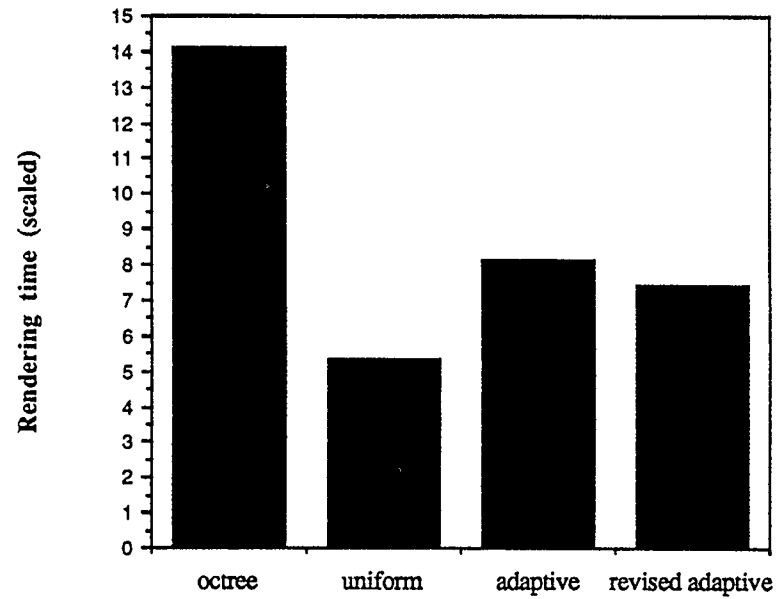
malised standard deviation of object distribution of 3.693 (section 3.2.2, figure 3.3). Because of this, the uniform subdivision algorithm outperforms the octree algorithm, although this performance difference is not dramatic. Octree subdivision allows faster ray traversal through the large empty areas under the leaves, but uniform subdivision performs better in the leafy parts of the tree, where the depth of the octree reaches five.

The adaptive voxel subdivision algorithm adapts to these two different areas of object density by subdividing the scene to a depth of two, and using uniform subdivision in the leafy parts of the tree. The adaptive algorithm performs best when using the first subdivision heuristic rather than the revised heuristic because the uniformity of object distribution favors a more uniform than adaptive subdivision.

### 5.5.3 Comparison of Performance for the Lumpy scene

The normalised standard deviation of object distribution for the **Lumpy** scene is 8.333 (section 3.2.2, figure 3.4), indicating that the scene is less uniform in object distribution than the **Tree** scene. This would suggest that an adaptive subdivision algorithm should outperform a uniform subdivision, as it can adjust to the variations in object density. As figure 5.30 shows, this is not the case.

The uniform subdivision algorithm outperforms the adaptive algorithms, and is almost three times faster than the octree algorithm. This is because only a small part of the scene, which extends for several city blocks around the street corner that is the focus of this image, is visible. The subset being viewed is much more uniform in object distribution than the entire scene, allowing the uniform algorithm to perform well.



no. rays	irreducible overhead time
976796	484 sec.

algorithm	subdivision	cpu time	scaled time	memory
octree	depth = 9	6829 sec.	14.12	2.6Mb
uniform	$g = 210^3$	2571 sec.	5.31	60.3Mb
adaptive	depth = 3	4015 sec.	8.30	2.8Mb
revised adaptive	depth = 4	3579 sec.	7.39	2.9Mb

Figure 5.30: Comparison of algorithm performance for the Lumpy scene.



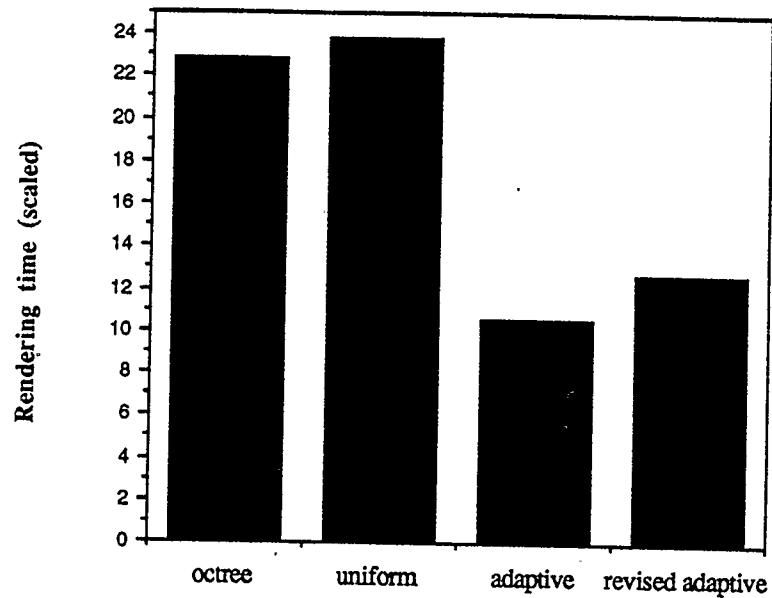
Adaptive voxel subdivision approaches the performance of uniform subdivision because it can adapt to areas of varying object density, yet can use uniform subdivision in areas of uniformity. The revised heuristic slightly outperforms the first heuristic, because it subdivides the root voxel more coarsely. If subdivision is coarse at the root and finer at deeper levels of the tree, as opposed to fine subdivision at the root and coarse subdivision in the children, there will be fewer child voxels, and less vertical traversal will be required to trace a ray through the scene.

While the uniform subdivision algorithm has the fastest rendering time, it requires more than twenty times the amount of memory that the adaptive voxel subdivision algorithm requires. If a hash table were used to represent only non-empty voxels, memory use would be greatly reduced, although ray traversal time would be increased.

#### 5.5.4 Comparison of Performance for the Drum scene

The normalised standard deviation of object distribution of 19.004 for the **Drum** scene (section 3.2.2, figure 3.5) indicates that the distribution of objects in the scene is irregular, and that an octree algorithm should outperform a uniform subdivision algorithm. As figure 5.31 shows, the uniform subdivision algorithm performs poorly, but the performance of the octree algorithm is only marginally better. The reason for the marginal performance of the octree algorithm is that the optimal depth of the subdivision hierarchy is seven, entailing much costly vertical traversal of rays.

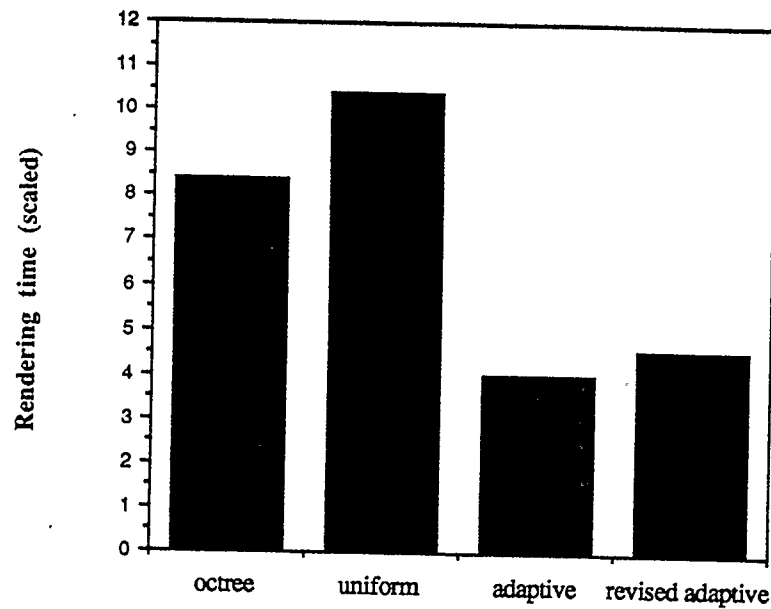
The adaptive voxel subdivision algorithm outperforms the octree and uniform algorithms by a factor of two. The first heuristic performs better than the revised heuristic because it reaches optimal subdivision at a depth of two, whereas the revised



no. rays	irreducible overhead time
578046	531 sec.

algorithm	subdivision	cpu time	scaled time	memory
octree	depth = 7	12116 sec.	22.82	64.4Mb
uniform	$g = 65^3$	12625 sec.	23.78	91.3Mb
adaptive	depth = 2	5622 sec.	10.59	86.4Mb
revised adaptive	depth = 3	6750 sec.	12.71	76.0Mb

Figure 5.31: Comparison of algorithm performance for the **Drum** scene.



no. rays	irreducible overhead time
491919	414 sec.

algorithm	subdivision	cpu time	scaled time	memory
octree	depth = 8	3481 sec.	8.41	29.3Mb
uniform	$g = 140^3$	4310 sec.	10.41	66.1Mb
adaptive	depth = 3	1669 sec.	4.03	38.7Mb
revised adaptive	depth = 4	1905 sec.	4.60	39.6Mb

Figure 5.32: Comparison of algorithm performance for the **Trike** scene.

heuristic requires a depth of three.

#### 5.5.5 Comparison of Performance for the **Trike** scene

The normalised standard deviation of object distribution for the **Trike** scene is 11.729 (section 3.2.2, figure 3.6). As the object distribution graph in figure 3.6 illustrates, most of the scene's volume is empty, and there are a small number of very dense voxels. This, combined with the fact that the entire scene is being viewed, rather than a subset of it, explains the octree algorithm's better performance com-

pared to the uniform subdivision algorithm. The ability of the octree to adapt to the extreme variations in object density offsets the vertical ray traversal entailed by an optimal octree depth of nine.

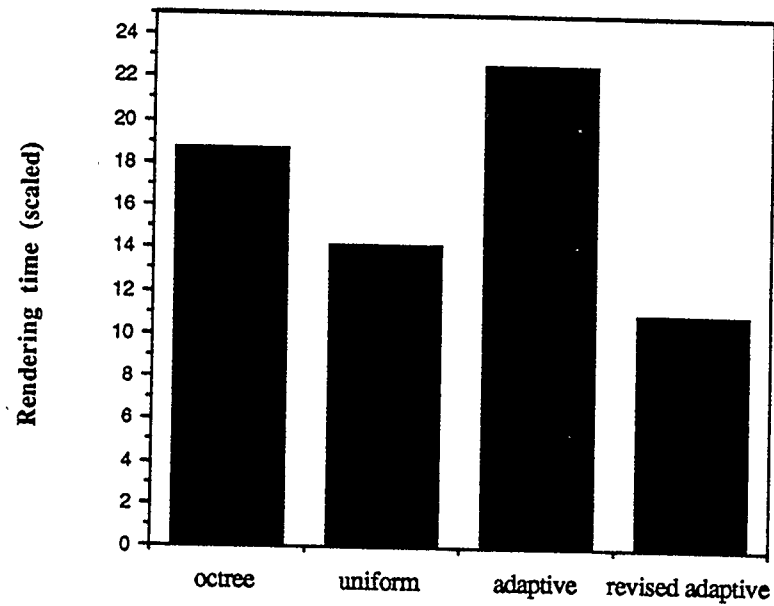
As with the **Drum** scene, the adaptive voxel subdivision algorithm outperforms both the uniform and octree subdivision algorithms by a factor of two. Again, the rendering time using the first heuristic is slightly faster than when the revised heuristic is used, due to its optimal tree depth of three versus an optimal tree depth of four for the revised heuristic.

As with the **Lumpy** scene, the memory requirements of the uniform subdivision algorithm could be brought into line with those of the adaptive algorithms by using a hash table to store only non-empty voxels.

#### 5.5.6 Comparison of Performance for the Car scene

The normalised standard deviation of object distribution of 11.337 for the **Car** scene (section 3.2.2, figure 3.7) would suggest a similar result as for the **Trike** scene. The graph of figure 3.7 (section 3.2.2) shows that the scene compositions are actually quite different. Much of the volume of the **Car** scene is composed of voxels containing a small number of polygons, whereas more than seventy-five percent of the **Trike** scene is empty. This explains the uniform subdivision algorithm's outperformance of the octree algorithm.

Adaptive voxel subdivision using the revised heuristic outperforms both the uniform and octree algorithms due to the prevalence of uniform object distribution combined with small areas of high object density. The first heuristic is expected to perform better than the revised heuristic, but this is not the case, because the



no. rays	irreducible overhead time
1768284	949 sec.

algorithm	subdivision	cpu time	scaled time	memory
octree	depth = 7	17691 sec.	18.64	8.9Mb
uniform	$g = 90^3$	13442 sec.	14.16	75.8Mb
adaptive	depth = 1	21447 sec.	22.60	23.8Mb
revised adaptive	depth = 3	10364 sec.	10.92	48.7Mb

Figure 5.33: Comparison of algorithm performance for the Car scene.

algorithm runs out of memory and can not subdivide beyond a depth of one, whereas the revised heuristic subdivides the scene to a depth of three without running out of memory.

## 5.6 Conclusions

The adaptive voxel subdivision algorithm outperforms uniform subdivision when rendering large complex scenes, because it can adapt to areas of varying object density, and uses uniform subdivision in areas of local uniformity.

Adaptive voxel subdivision is superior to octree subdivision for all types of scenes, because the depth of its subdivision hierarchies is smaller than for octree subdivision, so that fewer vertical traversals are performed when tracing a ray through a scene. The drawback of adaptive voxel subdivision as compared to octree subdivision is that the effects of over-subdivision manifest themselves more rapidly as the depth of the subdivision hierarchy is increased.

### 5.6.1 Adaptive Voxel Subdivision Heuristics

For most scenes, the first subdivision heuristic outperforms the more adaptive revised heuristic, because optimal subdivision hierarchy depths are smaller when using the first heuristic. The revised heuristic proves to be useful when rendering scenes composed of a large number of objects, such as the **Car**, because it coarsely subdivides dense voxels, thereby using less memory at each level of subdivision. Scenes that cause the algorithm to run out of memory when using the first heuristic can often be rendered efficiently by using the revised heuristic.

When rendering a subset of a scene, as in the **Lumpy** image, faster rendering times can be had by coarsely subdividing the root voxel, and subdividing more finely at lower levels of the subdivision hierarchy. Such a scheme reduces traversal in the root voxel, thereby reducing vertical ray traversal. A future direction for research is to develop a new subdivision heuristic that uses the revised heuristic of section 5.4.3 to subdivide the root voxel, and the first heuristic, described in section 5.4.1 at lower levels of the subdivision hierarchy.

### 5.6.2 The Effect of Scene Structure

The structure of a scene is inherently important in the performance of a subdivision algorithm. As the **Lumpy** scene illustrates, scene structure is not the only factor influencing the performance of a space subdivision algorithm. The areas of the scene through which rays are traced is highly important, because this defines the subset of the subdivision data structure that is traversed. The areas of the scene through which rays are traversed is defined by the position of the camera in the scene and the position of the lights.

The effect of rendering a subset of a scene is illustrated by the results of rendering the **Lumpy** scene in section 5.5.3. Although object distribution in the scene is not uniform, the subset of the scene that is in view is much more uniform than the scene as a whole. This allows a uniform subdivision algorithm to outperform the octree and adaptive voxel subdivision algorithms when rendering this image.

### 5.6.3 Ease of Use

The adaptive voxel subdivision algorithm does not provide optimal rendering times automatically, as unrestricting the maximum depth of the subdivision hierarchy causes over-subdivision, resulting in poor rendering times and excessive memory use. The optimal subdivision depth can be found for an image by rendering several test images at reduced resolution, and varying the maximum subdivision depth for each image. When rendering an animation sequence, a representative frame can be used to determine the optimal depth of the subdivision hierarchy for the sequence. Unless there are dramatic changes in the structure of the scene during the sequence, the optimal value of the depth parameter will not vary.

It is easier to find the optimal value of the subdivision parameter for the adaptive voxel subdivision algorithm than for the octree or uniform subdivision algorithms, because optimal values for the depth parameter usually lie between two and four for complex scenes. The optimal values of octree depth and uniform subdivision granularity for complex scenes have much more variance, requiring more test images to be rendered.



## Chapter 6

### Conclusion

A new space subdivision algorithm for ray tracing complex scenes has been presented. The algorithm, entitled *adaptive voxel subdivision*, integrates uniform and octree subdivision techniques, and outperforms both of these methods when rendering complex scenes.

The principal accomplishments of this thesis are:

- Introduction of the *object distribution graph* as a measure of scene structure.
- Analysis of a number of complex scenes, taken from animated films and commercial applications, showing that the distribution of objects in such scenes is not uniform, but that areas of local uniformity do occur.
- Development of the *adaptive voxel subdivision* algorithm and two heuristics for controlling subdivision.
- A tool for implementing space subdivision schemes.
- A comparison of the performance of octree, uniform, and adaptive voxel subdivision algorithms when rendering complex scenes.

#### 6.1 Future Work

A number of improvements that can be made to the adaptive voxel subdivision algorithm are presented in this section.

### 6.1.1 Controlling the Adaptive Voxel Subdivision

While the adaptive voxel subdivision algorithm performs well when rendering complex scenes, it requires the user to determine the optimal value for the maximum subdivision depth parameter. Cleary and Wyvill [Cleary 88] analysed the uniform subdivision algorithm and devised an equation to determine the optimal subdivision granularity for a scene, assuming a uniform object distribution and size. This approach can be extended to control the adaptive voxel subdivision of complex scenes.

A possible algorithm is to crudely subdivide voxels containing a large number of objects, and to measure the object density and size inside each of the crude subdivisions. Areas where the variance of these values is small are considered to be *meta-objects*. The number and size of these meta-objects can be used in the Cleary equations to determine the optimal subdivision granularity for the voxel. This process is repeated recursively until the variance of object density and size in a voxel is small, and the original Cleary and Wyvill equations are used to determine the subdivision granularity of the leaf voxel.

### 6.1.2 Optimisations to the Adaptive Voxel Subdivision Algorithm

#### Linking Neighboring Voxels

MacDonald and Booth [MacDonald 89] optimised the octree algorithm by linking adjacent voxels together, in order to reduce vertical tree traversal when tracing a ray through a scene. A link is made from each of the six faces of every leaf node in the octree to the smallest neighboring node whose voxel's surface completely encloses the face of the leaf in question. This significantly reduces vertical traversal during ray tracing, as a neighboring voxel is found by following a link pointer, rather than

ascending and descending the octree. MacDonald and Booth estimate that this technique reduces vertical traversal costs to between  $\frac{1}{4}$  and  $\frac{1}{7}$  of that incurred by Fujimoto's octree algorithm. This technique can be applied to the adaptive voxel subdivision algorithm.

### Lazy Initialisation

Charney and Scherson [Charney 90] proposed an optimisation to the traversal of  $k$ -d trees, where a ray descends to the bottom of the subdivision tree, and initialisation of horizontal traversal variables is performed only when the ray ascends the tree. This *interior node first* traversal algorithm could be combined with linked neighboring voxels to all but eliminate the costs of vertical traversal.

### Shared Traversal States

A third traversal optimisation was developed by Jevans [Jevans 90] to avoid duplicating the initialisation of horizontal traversal variables when a ray enters a voxel that is subdivided at the same granularity as one previously traversed. A table of *traversal states* is maintained, with one entry for every possible subdivision granularity. A traversal state stores the values of the variables used for horizontal traversal of a voxel subdivided at a given granularity. When a ray enters a voxel grid of the same granularity as one previously traversed, the traversal state table is accessed, and the values of the horizontal traversal variables from the previous traversal are used to initialise the traversal variables in the horizontal traversal routine.

This technique was implemented for a one dimensional subdivision scheme, but can be extended to the adaptive voxel subdivision algorithm

### Garbage Collection

The memory requirements of the adaptive voxel subdivision algorithm can be reduced by deleting subdivided voxels through which rays are no longer being traversed. When a voxel grid has not been accessed for some time, it can be deleted from the subdivision hierarchy, and the memory reused. If a ray passes through a voxel that has been collected, then the voxel must be resubdivided. Because adjacent rays exhibit spatial coherence, if pixels are rendered from the top of the screen to the bottom, it is unlikely that a collected voxel will be reaccessed and have to be resubdivided.

This technique would allow much larger scenes to be rendered than with the present implementation.

#### 6.1.3 A Hybrid Algorithm

As seen in chapter 5, over-subdivision in a hierarchical subdivision scheme results in poor performance. A hybrid algorithm, combining hierarchies of bounding volumes, ray classification, and adaptive voxel subdivision may be able to avoid the problem of over-subdivision.

One cause of over-subdivision is when the objects in a voxel are small and densely clustered (see figure 5.8, chapter 5). The adaptive voxel subdivision algorithm will recursively subdivide the voxel until it reduces the number of objects in the leaf voxels. This can result in an overly deep subdivision hierarchy, requiring costly vertical traversal when tracing rays. This could be avoided by surrounding clusters of small objects with bounding volumes, and continuing the adaptive voxel subdivision process inside the bounding volumes. Rays would be intersected with the bounding

volumes, rather than having to traverse through a number of recursive subdivisions.

The second cause of over-subdivision is when objects are large compared to the size of a voxel, and further subdivision puts them into all the child voxels (see figure 5.7, chapter 5). A simple solution is to avoid subdividing the voxel, although this results in a large number of ray-object intersection calculations for any ray that enters the voxel. An alternative is to use a ray classification scheme inside such a voxel to reduce the number of objects that have to be tested for intersection with a ray.

## 6.2 Conclusion

In chapter 5, the rendering times of the adaptive voxel subdivision algorithm are between four and eleven times slower than the irreducible overhead times of rendering the test images. This prompts the important observation that there remains less than an order of magnitude improvement possible for algorithms that are aimed at reducing the number of ray-object intersections required to trace a ray through a scene, indicating that future research in speeding up ray tracing must focus on reducing the number of rays required to trace an image. Possibilities include exploiting ray coherence for rendering single images, and frame coherence for rendering animation sequences.

## Bibliography

- [Ali87] Alias Research Inc. *ALIAS/1 User Guide*, 1987.
- [Allan 89] J. B. Allan, B. Wyvill, and I. H. Witten. A methodology for direct manipulation of polygon meshes. In *Proceedings of Computer Graphics International '89*, pages 451–469, 1989.
- [Amanatides 84] John Amanatides. Ray tracing with cones. *Computer Graphics*, 18(3):129–136, July 1984. Proceedings of ACM SIGGRAPH '84.
- [Amanatides 87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. *Proc. Eurographics '87*, 1987.
- [Appel 68] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conference*, number 32, pages 37–45, 1968.
- [Arnaldi 87] B. Arnaldi, T. Priol, and K. Bouatouch. A new space subdivision method for ray tracing csg modeled scenes. *The Visual Computer*, 3(2):98–108, 1987.
- [Arvo 87] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics*, 21(4), July 1987. Proceedings of ACM SIGGRAPH '87.
- [Arvo 90] J. Arvo and D. Kirk. Particle transport and image synthesis. *Computer Graphics*, 24(4), August 1990. Proceedings of ACM SIG-

## GRAPH '90.

- [Bartels 87] R. Bartels, B. Beatty, and B. Barsky. *An Introduction to Splines for Use in Computer Graphics & Geometric Modeling*. Morgan Kaufmann, Los Altos, California, 1987.
- [Blinn 86] James Blinn. The algebraic properties of homogeneous second order surfaces. *ACM SIGGRAPH Tutorial Notes*, 1986.
- [Charney 90] M. J. Charney and I. D. Scherson. Efficient traversal of well-behaved hierarchical trees of extents for ray-tracing complex scenes. *Visual Computer*, 6(3):167–178, June 1990.
- [Cleary 86] J. Cleary, B. Wyvill, G. Birtwistle, and Vatti R. Multiprocessor ray tracing. *Computer Graphics Forum*, 5(1):3–12, 1986.
- [Cleary 88] John Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Visual Computer*, 4:65–83, July 1988.
- [Cook 84] Cook, Porter, and Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):137–145, July 1984. Proceedings of ACM SIGGRAPH '84.
- [Dippé 84] Mark Dippé and John Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *Computer Graphics*, 18(3):149–158, July 1984. Proceedings of ACM SIGGRAPH '84.

- [Dippé 85] Mark Dippé. *Antialiasing in Computer Graphics*. PhD thesis, University of California, Berkeley, 1985.
- [Foley 90] Foley, vanDam, Feiner, and Hughes. *Computer Graphics: Principles and Practice (second edition)*. Addison-Wesley, 1990.
- [Fujimoto 86] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated ray-tracing system. *IEEE Computer Graphics & Applications*, 6(4):16–26, April 1986.
- [Glassner 84] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics & Applications*, pages 15–22, October 1984.
- [Glassner 89] A. Glassner. An overview of ray tracing. *An Introduction to Ray Tracing*, 1989. Edited by Andrew S. Glassner.
- [Goldsmith 87] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics & Applications*, pages 14–20, May 1987.
- [Haines 86] Eric A. Haines and Donald P. Greenberg. The light buffer: A shadow-testing accelerator. *IEEE Computer Graphics & Applications*, 6(9):6–16, September 1986.
- [Hall 83] Roy A. Hall and Donald P. Greenberg. A testbed for realistic image synthesis. *IEEE Computer Graphics & Applications*, 3(8):10–20, November 1983.



- [Hanrahan 86] Pat Hanrahan. Using caching and breadth-first search to speed up ray-tracing. In *Proceedings of Graphics Interface '86*, pages 56–61, 1986.
- [Heckbert 84] P. Heckbert and P. Hanrahan. Beam tracing polygonal objects. *Computer Graphics*, 18(3):119–127, July 1984. Proceedings of ACM SIGGRAPH '84.
- [Heckbert 89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, University of California (Berkeley), Computer Science Division, 1989.
- [Heckbert 90] P. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics*, 24(4):145–154, August 1990. Proceedings of ACM SIGGRAPH '90.
- [Jevans 89a] D. Jevans and M. Chmilar. Lumpy's Quest for Sev. 1989.
- [Jevans 89b] David A. J. Jevans. Optimistic multi-processor ray tracing. In *Proceedings of Computer Graphics International '89*, pages 507–552, 1989.
- [Jevans 89c] David A. J. Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89*, pages 164–172, 1989.
- [Jevans 90] David A. J. Jevans. Ray tracing scenes of varying local complexity. In *Proceedings of the Western Canadian Computer Graphics*

*Workshop '90, Banff, Alberta*, 1990. Also University of Calgary Research Report #90/378/02.

- [Joy 86] K. I. Joy and M. N. Bhetanabhotla. Ray tracing parametric surface patches utilizing numerical techniques and ray coherence. *Computer Graphics*, 20(4):279–285, 1986. Proceedings of ACM SIGGRAPH '86.
- [Kajiya 83] James T. Kajiya. New techniques for ray tracing procedurally defined objects. *Computer Graphics*, 17(3):91–102, 1983. Proceedings of ACM SIGGRAPH '83.
- [Kajiya 86] James T. Kajiya. The rendering equation. *Computer Graphics*, 20(4):143–150, 1986. Proceedings of ACM SIGGRAPH '86.
- [Kaplan 85] Michael R. Kaplan. The uses of spatial coherence in ray tracing. *SIGGRAPH '85 Course Notes 11*, 1985.
- [Kay 79] Timothy L. Kay. Transparency, refraction, and ray tracing for computer synthesized images. Master's thesis, Cornell University, January 1979.
- [Kay 86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, 1986. Proceedings of ACM SIGGRAPH '86.
- [Kingdon 86] Stewart Kingdon. Speeding up ray-scene intersections. Master's thesis, University of Waterloo, 1986.

- [Kobayashi 87] H. Kobayashi, T. Nakamura, and Y. Shigei. Parallel processing of an object space for image synthesis using ray tracing. *The Visual Computer*, 3(1):13–22, February 1987.
- [Kunii 85] T. L. Kunii and G. Wyvill. A simple but systematic CSG system. In *Proceedings of Graphics Interface '85*, pages 329–335, 1985.
- [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.
- [MAGI 68] (Mathematical Applications Group Inc.) MAGI. 3-d simulated graphics. *Datamation*, February 1968.
- [Mitchell 87] Don P. Mitchell. Generating antialiasing images at low sampling densities. *Computer Graphics*, 21(4):65–72, July 1987. Proceedings of ACM SIGGRAPH '87.
- [Nemoto 86] K. Nemoto and T. Omachi. An adaptive subdivision by sliding boundary surfaces for fast ray tracing. In *Proceedings of Graphics Interface '86*, pages 43–48, 1986.
- [Nishimura 83] H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa, and K. Omura. Links-1: A parallel pipelined multimicrocomputer system for image creation. *IEEE 1983 Conference Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983.

- [Ohta 87] M. Ohta and M. Maekawa. Ray coherence and constant time ray tracing algorithm. In *Proceedings of Computer Graphics International '87*, pages 303–314, 1987.
- [Pearce 87] Andrew Pearce. An implementation of ray tracing using multiprocessing and spatial subdivision. Master's thesis, University of Calgary, Dept. of Computer Science, 1987.
- [Roth 82] S. D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, (18):109–144, 1982.
- [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. Proceedings of ACM SIGGRAPH '80.
- [Samet 84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.
- [Scherson 87] L. Scherson and E. Caspary. Data structures and the time complexity of ray tracing. *The Visual Computer*, 3(4):201–213, 1987.
- [Scherson 88] I. D. Scherson and E. Caspary. Multiprocessing for ray-tracing: A hierarchical self-balancing approach. *Visual Computer*, 4(4):188–196, October 1988.
- [Shinya 87] M. Shinya, T. Takahashi, and N. Seiichiro. Principles and applications of pencil tracing. *Computer Graphics*, 21(4):45–54, July 1987. Proceedings of ACM SIGGRAPH '87.

- [Sillion 89] François Sillion and Claude Puech. A general two-pass method integrating specular and diffuse reflection. *Computer Graphics*, 23(3):335–344, 1989. Proceedings of ACM SIGGRAPH '89.
- [Snyder 87] John M. Snyder and Alan H. Barr. Ray tracing complex models with surface tessellations. *Computer Graphics*, 21(4):119–128, July 1987. Proceedings of ACM SIGGRAPH '87.
- [Speer 85] L. Richard Speer, Tony D. DeRose, and Brian A. Barsky. A theoretical and empirical analysis of coherent ray-tracing. In *Proceedings of Graphics Interface '85*, pages 1–8, 1985.
- [Sutherland 74] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, March 1974.
- [Vatti 85] Reddy Vatti. Multiprocessor ray tracing. Master's thesis, University of Calgary, Dept. of Computer Science, 1985.
- [VonHerzen 87] Brian VonHerzen and Alan H. Barr. Accurate triangulations of deformed intersecting surfaces. *Computer Graphics*, 21(4):103–110, July 1987. Proceedings of ACM SIGGRAPH '87.
- [Wallace 87] John R. Wallace, Michael F. Cohen, and Donald P. Greenberg. A two-pass solution to the rendering equation. *Computer Graphics*, 21(4):311–320, July 1987. Proceedings of ACM SIGGRAPH '87.

- [Wallace 89] John R. Wallace, Kells A. Elmquist, and Eric A. Haines. A ray tracing algorithm for progressive radiosity. *Computer Graphics*, 23(3):315–324, 1989. Proceedings of ACM SIGGRAPH '89.
- [Ward 88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. *Computer Graphics*, 22(4):85–92, 1988. Proceedings of ACM SIGGRAPH '88.
- [Weghorst 84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.
- [Whitted 80] Turner Whitted. An improved illumination model for shaded display. *Comm. ACM*, 23(6):343–349, June 1980.
- [Woo 90] A. Woo and J. Amanatides. Voxel occlusion testing: A shadow determination accelerator for ray tracing. In *Proceedings of Graphics Interface '90*, pages 213–220, 1990.
- [Wyvill 85] G. Wyvill and T. L. Kunii. A functional model for constructive solid geometry. *The Visual Computer*, 1(1):3–14, 1985.
- [Wyvill 86a] Brian Wyvill, Craig McPheeters, and Rick Garbutt. The University of Calgary 3D Computer Animation System. *Journal of the Society of Motion Picture and Television Engineers*, 95(6):629–636, 1986.

- [Wyvill 86b] Brian Wyvill, Craig McPheeters, and Geoff Wyvill. Animating soft objects. *The Visual Computer*, 2(4), February 1986.
- [Wyvill 86c] G. Wyvill, T. L. Kunii, and Y. Shirai. Space division for ray tracing in CSG. *IEEE Computer Graphics & Applications*, 6(4):28–34, 1986.
- [Wyvill 86d] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4), February 1986.