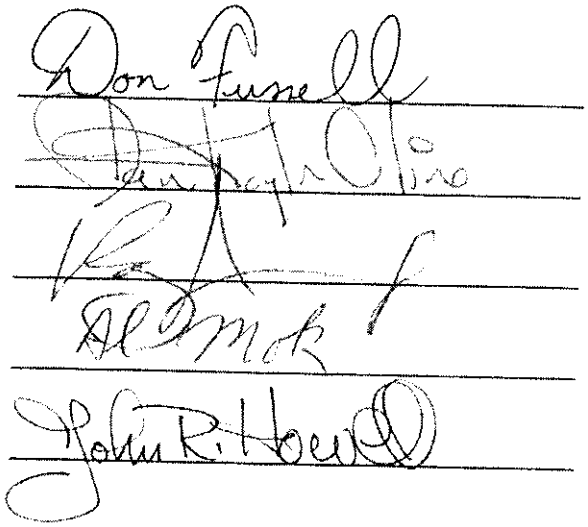

SEAL OF THE UNIVERSITY OF TEXAS
CIVITATIS
DISCIPLINA
PRAESIDIVM

ADAPTING SEARCH STRUCTURES TO SCENE CHARACTERISTICS FOR RAY TRACING

APPROVED BY

SUPERVISORY COMMITTEE:



To my wonderful parents,
Kalpathi S. Raman and Lakshmi Raman

ADAPTING SEARCH STRUCTURES TO SCENE CHARACTERISTICS FOR RAY TRACING

by

KALPATHI RAMAN SUBRAMANIAN, B.E., MSCS

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1990

Acknowledgments

First and foremost, I would like to thank my parents for their exceptional support and encouragement in my educational pursuits. I would also like to thank my brothers, especially Dr. K.R. Seshan and my sisters, for their support over the years.

I would like to deeply thank my advisor, supervisor and good friend, Don Fussell, from whom I have learned a great deal. He was especially helpful in getting me started on the subject of this dissertation, and, all through the years forcing me to look at the problem from several different angles. He always let me work at my own pace, thus allowing me to make smooth progress in my research.

I would like to thank my committee members Alan Cline, Roy Jenevein, John Howell and Al Mok for their active participation. They have been especially responsible in going through the dissertation very carefully and forcing me to make several revisions, thus resulting in a high quality document. They have also sharpened my writing and organizational skills, to which I am deeply grateful, since this will be very useful in the years to come.

I sincerely appreciate and thank the Computer Science department of the University of Texas at Austin for accepting me into their doctoral program, failing which none of this would have happened. I feel privileged to be a student of this department. I am very grateful for the financial support of the department. I would like to thank all the people in the department who have

been helpful to me over the years, especially Gloria Ramirez. I am also very grateful for the fine computational and laboratory facilities extended to me by this department for performing this work.

I would like to thank my colleagues in the graphics group Gordon Fossum, A. T. Campbell, Chris Buckalew, Kelvin Thompson, Don Speray, Jesse Driver, Bob Swaffar and Alita Rumsey for constructive criticisms, technical discussions and suggestions. The presence of a group working in the same area was very helpful for communicating and testing out new ideas. This has definitely impacted my research. In addition, thanks to my theory friends, Sampath Rangarajan, Ramki Thurimella and Nandit Soparkar for help with some parts of this work. Thanks also to Jim Meyering for a variety of things associated with software tools.

Thanks are also due to Dr. James Almond, Dr. Matthew Witten and the Graphics and Vision group, especially Jesse Driver, at the Center for High Performance Computing. I would like to thank Dr. Almond for giving me an opportunity to be a part of the graphics group at the Center for High Performance Computing. This has directly resulted in the chapter on volume rendering. This opportunity has been responsible for my new interest in scientific visualization and also contributed to my first publication in an important conference on visualization.

KALPATHI RAMAN SUBRAMANIAN

The University of Texas at Austin

December, 1990

ADAPTING SEARCH STRUCTURES
TO SCENE CHARACTERISTICS
FOR RAY TRACING

Publication No._____

KALPATHI RAMAN SUBRAMANIAN, Ph.D.
The University of Texas at Austin, 2023

Supervising Professor: Donald S. Fussell

Ray tracing is an important and popular rendering technique in computer graphics for synthesizing photorealistic images. However, ray tracing, if not carefully done, can be a computationally expensive technique. A great deal of research has focused on discovering efficient ways to perform ray tracing. An important approach to controlling the computational expense has been the use of geometric search structures to prevent needless ray-object intersection calculations.

Search structures in current use take advantage of scene characteristics in a variety of ways to enhance ray tracing performance. Constraints in their construction can cause inefficiencies and consequent degradation in performance. Performance comparisons between search structures using timing benchmarks have shown that no single existing search structure performs best on all scene models. A knowledge of search structure performance prior to rendering is therefore important to selecting a search structure for a given scene.

A thorough understanding of the ways in which search structures succeed in enhancing performance on various types of scenes can also be expected to lead to improvements in existing techniques.

We present new results in adapting search structures to scene characteristics for improving the performance of ray tracing. A cost model is developed for evaluating search structures currently being used in ray tracing. The model has been successfully used to terminate search structure construction, thus making it unnecessary to set termination parameters in advance. The model has also been used with limited success to compare the performance of different search structures for a given scene.

A detailed experimental study of some of the important properties of search structures has been performed. This has resulted in a new adaptive search structure that is based on k-d trees, a multi-dimensional binary search structure which outperforms existing methods. Its high performance is primarily due to the fact that it combines the advantages of such structures based on space partitioning and those based on bounding volumes. The greater flexibility of this search structure allows it to terminate automatically at a point where further subdivision would result in no additional benefits.

Finally, this search structure has been used to render volume models from scientific applications such as medical imaging and molecular modeling. Its advantages over traditional volume rendering techniques have been demonstrated.

Table of Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

In recent years, ray tracing has become an important rendering technique for synthesizing photo-realistic images in computer graphics. Ray tracing has the capability of generating lighting effects such as gloss, translucency, motion blur and penumbras [10]. The ease of implementing ray tracing has made it a very popular method for a variety of applications. However, ray tracing, if not carefully done, can be a computationally expensive technique. Consequently a great deal of research has focused on discovering efficient ways to perform ray tracing.

The principal expense in ray tracing lies in determining a ray's first (i.e., closest to the ray's origin) intersection with an object in a scene. This must be done several times per pixel, the exact number depending on the effects being generated and the scene being rendered. Research on efficient algorithms has quite properly focused on minimizing the cost of these intersection calculations. Bounding volumes [26][39][54][55], space partitioning structures [15][6][17][25][43], item buffers [54], shadow buffers [21] and ray coherence techniques [1][23][44] all have proven effective at improving the efficiency of ray tracing. Each of these methods use some form of a 'search structure' to organize the object primitives in the input scene. A search structure is simply a data structure that has enough information in it to provide efficient access to a particular data item without having to look at the entire data collection.

All these methods are being used with great success. However, performance evaluations of each technique until now have been achieved solely via timing benchmarks. Comparisons between different methods have also been difficult since each method was tested by researchers on their own set of benchmarks, thus precluding any meaningful results. A first step has been taken by Haines [20] to rectify this situation, in proposing a set of test scenes to be used as benchmarks for measuring performance. Detailed specifications of the viewing and lighting parameters as well as surface characteristics of the scenes have been provided in his proposal. If two methods use the same test case with identical parameter values, their rendering times are a true measure of their respective performances. This is because the total number of rays spawned will be the same in both cases (identical viewing parameters, image resolution and rules on when to terminate tracing each ray ensure this). Both methods have to trace the same number of rays and the method which does this more efficiently will prevail.

Even this is inadequate. Some search structures, while performing well on one scene, might perform poorly on another with totally different characteristics. In this case, knowledge of the performance of a search structure is required before rendering the scene so that a decision may be made to use it or substitute it with another.

The search structures that are currently in use have a variety of constraints in their construction. For example, in the octree and BSP hierarchies that are used for ray tracing, partitioning planes are located centrally within the scene extents at every stage of the subdivision. Also the extents are simultaneously subdivided along all three dimensions. Since the input scene is accessed through the search structure, how do these constraints affect perfor-

mance? This is a question that is not very easily or very well answered.

In this dissertation, we study the properties of search structures that are currently being used to accelerate ray tracing. We are particularly interested in how well they adapt themselves to the characteristics of a scene. This will give us an idea of the strengths and weaknesses of each structure and provide us a better understanding of their performance characteristics on any scene. Once we achieve this, we will use this information to design new search structures that are superior to current search structures.

We begin by developing a cost model for search structures used in ray tracing. The cost model uses statistical characteristics of the search structure (and hence, the underlying scene, since the scene distribution influences the construction of most search structures) in arriving at an estimate for the cost of tracing a ray for a particular scene. The model provides two important optimizations to ray tracing. First, the cost model can be evaluated as the search structure is constructed. So, an estimate of the cost of tracing each ray is known prior to rendering. Since the construction of the search structure usually takes a small fraction of the rendering time, it is possible to construct several search structures, compare their costs (for a particular scene) and choose the one with the smallest cost. We demonstrate this application in chapter 4.

The second application of the cost model deals with the termination problem of search structures used in ray tracing. A long standing problem in the construction of a search structure has been the determination of the correct depth at which it should be terminated. In order to optimize performance, this threshold must be set correctly. If it is too low, then large numbers of object primitives end up in each region; if it is too high, getting to the leaf nodes of the search structure is more expensive.

One of the parameters of the cost model is the depth (or amount of subdivision, to be more general) of the search structure in question. The reported cost varies with the depth of the search structure. To determine the correct depth, the cost model is evaluated each time the depth of the search structure is increased. The correct point to terminate the search structure is the point at which the cost becomes a minimum. We have found the model to be a very effective means to solve the termination problem for search structures such as the octree, BSP tree, the uniform subdivision method and the automatic bounding volume hierarchy.

Our next step has been an experimental study of some of the common properties of search structures currently being used in ray tracing. Characteristics such as locations and orientations of partitioning planes, effects of using simple volumes to surround sections of the scene (like a sphere or a rectangular parallelepiped), presence of void areas (void areas refer to sections of the environment where there are no object primitives) at the nodes of search structures and hierarchy traversal methods are studied to determine their effect on performance. We have implemented a testbed system where each of these characteristics can be studied individually or in combination to see their effect on performance.

The net result of this study has led to the the development of a new search structure which falls under the class of k-d trees, which are multidimensional binary structures introduced by [2][3]. The k-d tree, that has been adapted to ray tracing, combines the advantageous characteristics of space-partitioning structures such as the octree and BSP trees as well as those based on hierarchies of bounding volumes. At the same time, it has greater flexibility in its construction, allowing better adaptability to scene characteristics. Ex-

perimental results demonstrate the superior performance of the *k-d* tree over search structures currently used in ray tracing. We validate these results with the cost model that we described earlier. Finally, the greater flexibility in the construction of the *k-d* tree (specifically, in locating partitioning planes) helps it to terminate automatically at the point where further subdivision would result in no additional benefits.

The dissertation is organized as follows. Chapter 2 describes the fundamentals of ray tracing. Chapter 3 describes some of the most commonly used search structures for ray tracing, chapters 4 and 5 describe the cost model and its applications, Chapter 6 analyzes the important characteristics that affect performance of search structures used in ray tracing and describes in detail the construction and use of the *k-d* tree. Chapter 7 describes the use of the *k-d* tree in rendering volume models for scientific applications. In chapter 8, we present our conclusions and future work in this area. Chapter 9 contains color plates of images used in the implementation.

Chapter 2

RAY TRACING

We begin by describing the fundamentals of ray tracing in computer graphics. We will describe the basic ray tracing algorithm in the context of rendering and then talk about extensions of the basic algorithm that solve some important problems as well as generate special effects. This will help us understand the computations involved in ray tracing and potential for speedups.

2.1 Rendering

In rendering, we are concerned with producing realistic images of models containing graphical objects. Graphical objects could be as simple as points or vectors, or more complicated surfaces such as polygons, higher order surfaces such as B Spline surfaces or even solid objects. Creation of realistic images involves modeling the interaction of light energy with the objects in the environment. The more accurate this is, the more realistic the images will be. An equally important factor is visual perception. A good understanding of the visual perception of the human eye is important to determining the intensities that are actually sensed from an environment. Let us begin by describing the problem we are trying to solve in realistic rendering.

The intensity I of the reflected light at a point on a surface is given

by the following double integral:

$$I(\lambda, \phi_i, \theta_i, \phi_r, \theta_r) = \int_{\phi_i} \int_{\theta_i} L(\lambda, \phi_i, \theta_i) R(\lambda, \phi_i, \theta_i, \phi_r, \theta_r) d\phi_i d\theta_i \tag{2.1}$$

Here light energy is incident on a surface from the direction (ϕ_i, θ_i). Intensity I measures part of the reflected intensity along (ϕ_r, θ_r). Energy of incident light (provided by the function L) is expressed per unit time and per unit area of the reflecting surface, while intensity is measured per unit projected area, and, per unit of the solid angle. R is the bidirectional reflection function, which means that R is symmetric with respect to incidence and reflection angles. $I, L, andR$ are all dependent on the wavelength of light, λ. Ideally, a separate intensity needs to be computed for each wavelength and these intensities must be mapped into a color space that can be displayed on a workstation. It is more common (for computational efficiency) to just compute intensities for the red, green and blue color bands. We will adopt this simplification when we start developing a lighting model for ray tracing.

In cases where there are objects that transmit or emit light, the reflection function can be replaced by a transmittance function or an emission function. If we can determine the intensity at any point in the environment, then the next step is to determine what fraction of it arrives at the eye. Before we get into this, we need to define how an image is created on a screen.

2.1.1 The Camera Model

Generating three dimensional images for display on a computer screen is similar to recording a scene onto film with a camera. An understanding of how a camera works is useful.

Perhaps the most common and simple camera model is the *pinhole camera*, as illustrated in Fig. 2.1.

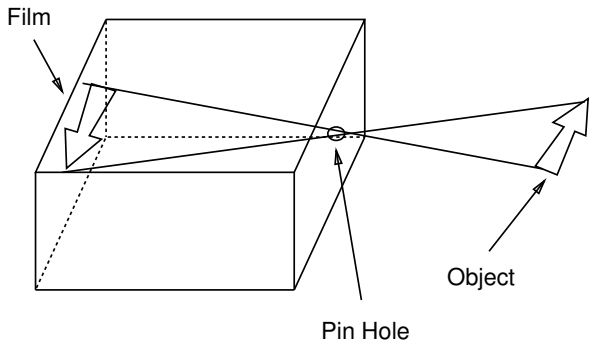


Figure 2.1: The Pin-Hole Camera Model.

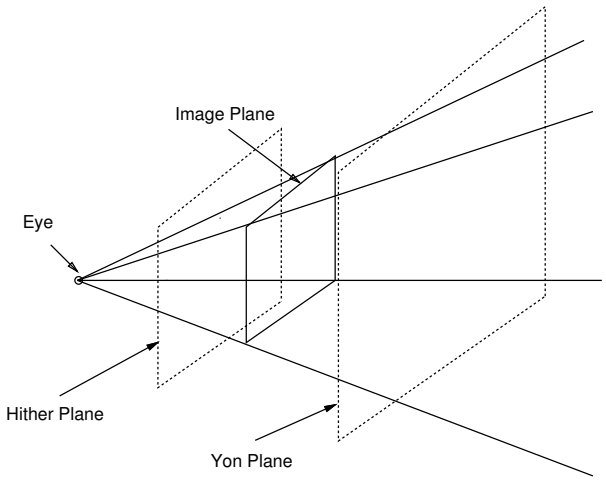


Figure 2.2: The Viewing Frustum.

In Fig. 2.1 light rays enter the pinhole from all directions and strike the film at different points to contribute to the image. The pinhole serves to restrict the amount of light that is received by the film, by making sure that each point is illuminated by light energy only along the line joining that point to the pinhole.

In computer graphics, it is customary to position the eye at the pinhole and move the film out in front of the pinhole. The pinhole becomes the eye and the plane of the film becomes the image plane or screen. Just as in the pinhole camera, all light rays in the computer graphics model are required to

pass through the eye. Creation of an image involves projecting the scene within the *viewing frustum* (the truncated pyramid in Fig. 2.2) onto the image (or projection) plane. The rectangular window shown in Fig. 2.2 is where the image will be formed. Usually, two clipping planes called the *hither* and *yon* planes are placed parallel to the image plane. Only objects between these planes and contained by the four walls of the truncated pyramid are projected onto the image plane. This is to to avoid large objects close to the view point blocking the rest of the environment. Also, graphics workstations which contain hardware for performing hidden surface removal using the z-buffer algorithm have a limited word size for storing z depth values. Limiting the range along the line of sight using the hither and yon planes helps increase the precision and avoids making wrong decisions on determining the closest object seen at each pixel.

Creating an image involves sending out rays from the eye through points in the image plane into the environment and determining their intensity. However, there are an infinite number of points in the image plane. Instead, it is more practical to divide up the image plane into rectangular regions and determine the average intensity of each region. The number of regions into which the image plane is divided is the *resolution* of the image. Each region is called a *pixel*. Each pixel in the image plane represents a window into the environment. The problem now is reduced to determining an accurate color for each pixel in the image plane. Much of the research in computer graphics rendering has gone into developing sophisticated techniques to answer this question.

2.2 Local Illumination Models

Solving the double integral in Equation 2.1 analytically is very difficult. What we would like is an approximation to this reflection integral.

The earliest lighting models used to generate images made several simplifying assumptions for computational efficiency. They include the following:

- Only illumination from designated light sources in the environment was considered. This means that light reflected from a point towards the eye is only due to the reflection taking place at that point from these light sources in the environment. Illumination arriving at this point through reflection from other objects in the environment is ignored or approximated by a constant.
- The intensity at a point was computed only at three different wavelengths, usually red, green and blue.
- The reflection function used is usually a constant for each wavelength.

The *local* reflected light usually consists of three parts, those due to diffuse, specular and ambient reflection.

2.2.1 Diffuse Reflection

In diffuse surfaces, light, after striking a surface, is scattered equally in all directions. Diffuse light can be considered as light that has penetrated the surface of an object, been absorbed, and then re-emitted. This is common in rough surfaces. The red component of the intensity of light reflected by a perfect diffuser is given by Lambert's cosine law:

$$I_r = I_{lr} k_{dr} cos\theta, \qquad 0 \leq k_{dr} \leq 1 \tag{2.2}$$

where I_r is the red component of the reflected intensity at a point on the surface, I_{lr} is the red component of the incident intensity from a point light

source, k_dr is the red component of the diffuse reflection constant and θ, the angle between the vector to the light source and the surface normal at the point. Similar equations can be written for green and blue bands.

2.2.2 Ambient Reflection

Objects modeled with the Lambertian model appear dull and points that do not receive light from the point sources appear black. In reality, objects receive light scattered back from their surroundings. As a first step, this component of reflection is approximated by a constant term. This *ambient* light is combined with the diffuse component as follows:

$$I_r = I_{ar}k_{ar} + I_{lr}k_{dr}cos\theta \tag{2.3}$$

where I_{ar} is the red component of the ambient light intensity and k_{ar} is the red component of the ambient reflection constant ($0 \leq k_{ar} \leq 1$), used to indicate how much of the ambient light is reflected from the surface, ($0 \leq k_{ar} \leq 1$).

2.2.3 Attenuation for Distant Objects

If we use the above model to compute the intensity of light from two objects identically oriented but at different distances from the eye, we obtain the same intensity. If their projections overlap, then it is not possible to distinguish between them. It is well known [22] that light energy decreases inversely with the distance from the light source (and hence, intensity varies inversely with the square of the distance). However, if the light source is assumed to be at infinity, there is no contribution from the diffuse term. $1/d$ attenuation was used by Warnock [53] and $(1/d)^x$ was used by Romney [37]. Warnock was trying to account for attenuation due to atmospheric fog whereas Romney's function was intended to obey the inverse square law for energy

from a point source. results have shown that $x = 4$ produces more realistic results. The justification for using these functions instead of the inverse square function suggested by theory is that the models are being applied using shading techniques that cannot capture the information required for theoretically based models. Global illumination models such as those used by the radiosity method or distributed ray tracing handle light attenuation accurately since they balance the flow of light energy within the environment.

For our simple lighting model, we will assume an inverse distance function to model attenuation of light intensity reaching the eye.

Our lighting model becomes

$$I_r = I_{ar}k_{ar} + \frac{I_{lr}}{d+K}k_{dr}cos\theta \tag{2.4}$$

where K is an arbitrary constant.

2.2.4 Specular Reflection

Specular reflection occurs from smooth surfaces such as mirrors. The intensity of specularly reflected light depends on angle of incidence, the wavelength of light and surface properties. The governing equation is the Fresnel equation. For perfect specular reflection, Snell's law holds, stating that the angle of incidence equals the angle of reflection. Only an observer stationed along the reflection vector R sees any specularly reflected light, as in Fig. 2.3. For imperfectly reflecting surfaces, the amount of light reaching an observer depends on the spatial distribution of the specularly reflected light.

Highlights on shiny objects are due to specular reflection. Since specular reflection is concentrated around the reflection vector, highlights move as the observer moves. Also, specularly reflected light exhibits the characteristics

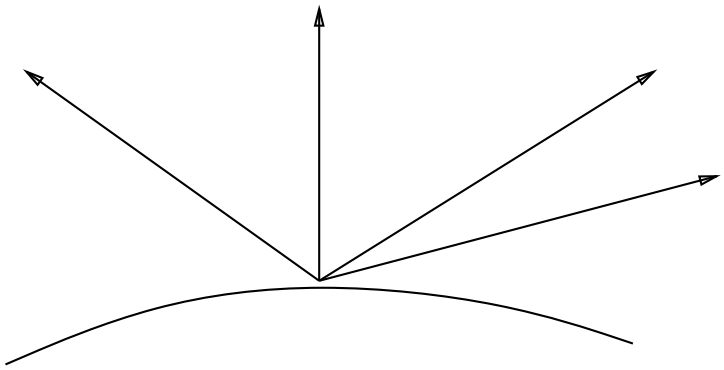


Figure 2.3: Specular Reflection

of incident light. Thus, highlights on a "blue" surface from a white light source are white, rather than blue.

An empirical model for specular reflection is given by Phong [36],

$$I_{sr} = I_{lr} w(i, \lambda) cos^n \alpha \tag{2.5}$$

where I_{sr} is the intensity of specularly reflected light. $w(i, \lambda)$ is a reflection function, giving the ratio of specularly reflected light to the incident light as a function of the incidence angle and the wavelength of light. n is a power that approximates the spatial distribution of the specularly reflected light. Higher values of n yield focused spatial distribution characteristics of metals and other shiny surfaces, while small values of n yield more distributed results, for example, in nonmetallic surfaces. α is the angle between the reflected ray and the line of sight, as shown in Fig. 2.3.

Including specular reflection, our lighting model becomes

$$I_r = I_{ar} k_{ar} + \frac{I_{lr}}{d + K} (k_{dr} cos\theta + I_{lr} w(i, \lambda) cos^n \alpha) \tag{2.6}$$

where d is the distance from the viewpoint to the object and K is an arbitrary constant. In practice, $w(i, \lambda)$ is replaced by a constant k_{sr} $(0 \leq k_{sr} \leq 1)$, where

k_{sr} is the specular reflection constant for the red band. This yields

$$I_r = I_{ar}k_{ar} + \frac{I_{lr}}{d+K}(k_{dr}cos\theta + I_{lr}k_{sr}cos^n\alpha) \tag{2.7}$$

In the presence of m light sources, the lighting model becomes

$$I_r = I_{ar}k_{ar} + \sum_{j=1}^{m} \frac{I_{lr_j}}{d+K}(k_{dr}cos(\theta_j) + k_{sr}cos^n\alpha_j) \tag{2.8}$$

and similarly for green and blue bands.

2.3 Global Illumination Models

Equation 2.8 is an example of a *local* lighting model. Only lighting from designated light sources are taken into account in this model. It does not consider illumination from other objects in the environment (except for light sources). As we saw earlier, this illumination was lumped into the constant ambient light. For synthesizing realistic images, this indirect illumination component must be computed with more accuracy. In global illumination models, the light that reaches a point by reflection from, or transmission through, other objects in the scene, as well as light incident from light sources is considered in determining the intensity of the light reflected from a surface point to the observer.

Two important techniques that compute global illumination are *ray tracing* [55][38] and radiosity [19][8][50][7].

2.3.1 Ray Tracing

Going back to Equation 2.1, we mentioned that an approximation to this integral is necessary. The local lighting model we developed in the previous section is a good approximation for illumination received from designated light

sources in the environment. However, a point in the environment can also receive illumination from another object through reflection from the object (or transmission through the object). To model this indirect illumination, we need to consider light coming at any surface point from points on neighboring objects, and light coming to these neighboring points from their neighboring points and so on. Since there could potentially be a large number of objects in the environment, and light could arrive from any direction that contains a surface point, this makes the problem of determining global illumination computationally infeasible.

Traditional ray tracing makes several simplifications.

1. The surfaces in the environment are perfect reflectors. In this case, the surface acts like a mirror and reflects light only along the mirror direction (the angle of incidence equals the angle of reflection). This causes sharp reflections.

   In practice, surfaces are rarely perfect reflectors or transmitters. Imperfections in the surface could cause light to scatter. This illumination has been modeled using different distribution functions. For instance, both Gaussian and Beckmann functions have been used as microfacet distribution functions to estimate the reflection of light from a surface, which is modeled as a collection microfacets.

   In an effort to reduce computation cost, a simple approximation to this is used. The cosine power function that we used to model specular highlights from point sources (described in the previous section) is used to account for this component of the reflected illumination. This function tends to conentrate the reflection illumination around the mirror reflection direction.

Thus, surfaces in the environment are perfect reflectors or transmitters as far as other surfaces in the environment are concerned, however, they are not so as far as the designated light sources in the environment are concerned.

2. If a surface is transparent, transmission is only through the refraction direction, which depends on the angle of incidence and the refractive index of the material of the surface.

3. All light sources in the environments are point light sources. To determine if a surface point is in shadow, a ray is spawned to each point source and tested to see if it is blocked by any object in the environment. Approximating light sources by points results in sharp shadows.

4. All other illumination (for example, light arriving at a point due to diffuse reflection from another point in the environment) is approximated by a constant ambient light source.

5. In addition, if a surface point is not in shadow, then a local lighting model like the one we developed earlier will be used to approximate direct illumination from the light sources in the environment.

These assumptions drastically reduce the number of rays that need to be spawned and processed to determine global illumination. The main difference in the lighting model proposed for use in traditional ray tracing is the illumination that is received from neighboring points through the mirror reflection and transmission directions.

Later on, we will see how some of these assumptions can be relaxed at the expense of additional rays, to obtain greater photo-realism. We next describe the traditional ray tracing algorithm.

2.3.1.1 Forward and Backward Ray Tracing

With the above assumptions in our mind, the problem is to determine the color of each pixel as seen from the eye. The question is, 'What amount of light passes through the center of each pixel ending at the eye ?' This gives us a ray definition: one of the endpoints is the eye point and its direction is defined by the vector from the center of a pixel to the eye.

Light rays originate from light sources with intensities and in directions determined by their characteristics. Each ray starts with a certain intensity and direction and may hit some object in the scene. If it does not hit any object, then it travels through open space until its intensity reduces to zero. If it does strike an object, several things happen. A portion of it is absorbed by the object, another portion is reflected, and if the surface is transparent, some of it is also transmitted. In general, light is also scattered, but conventional ray tracing does not model this. The reflected and transmitted rays are followed individually, and they may hit other objects in the scene, leading to more reflection and transmission. This process continues until each ray's energy dies out.

A fraction of all these rays will hit the image plane and end up at the eye. These are the intensities we are interested in, since these are exactly the rays that start from the eye and sample the environment. However, most of the rays do not reach the eye and are useless for image synthesis.

What we have just described is *forward ray tracing*. We cannot simulate this process directly since this involves tracing potentially an infinite number of rays, most of which is wasteful. However, we know exactly the rays we are interested in: those that pass through the center of each pixel and end

up at the eye. If we reverse the ray tracing process, starting from the eye position and passing through the center of each pixel, then we are tracing rays backward, and this is termed *backward ray tracing* or just *ray tracing*. In this process we start with the ray connecting the eye and the center of the pixel in question and extend this into the environment and test for an intersection with any object in the scene. The first (closest) hit is what we are interested in, since this is the point from which light travels to the eye through the center of the pixel. Now this point could have obtained its illumination from several sources: directly from the light source(s), by emission (in which case the surface emission characteristics need to be known to compute this component of light), by reflection, of incident light at this point into the eye, and lastly by transmission (if the object is transparent) through this point into the eye. There are other ways light could reach the eye, notably diffuse reflection from nearby objects, but, as stated earlier, the reflection function R models only pure specular reflection and transmission.

Thus, starting with the ray through the pixel in question, we have spawned several new rays, towards the light source(s), in the reflection direction and in the transmission direction. Rays are also spawned toward each light source from a surface point to determine if it is in shadow. Points in shadow do not receive any light energy from the light source in question. In Equation 2.8, this means that the diffuse and specular terms will be zero for that particular light source. Now treating the intersection point at which these rays are spawned as a new ray origin, we trace these rays and determine their intensities. As can be expected, this leads to a recursive algorithm [55]. Once the intensities from these different sources are determined, they are weighted suitably depending on the surface properties of the object and added. Essen-

tially, a local lighting model like the one in Equation 2.8 is applied at each intersection point to determine the total intensity at that intersection point. Once all intensities are computed then we have the color of the pixel. The same process is applied to every pixel in the scene.

This recursive ray tracing algorithm was proposed by Whitted [55]. Whitted's global lighting model is the same as Equation 2.8 with the addition of two more terms accounting for indirect specular reflection and transmission. The red component of the intensity is given by

$$I_r = I_{ar}k_{ar} + \frac{I_{lr}}{d+K}(k_{dr}cos(\theta) + I_{lr}k_{sr}cos^n(\alpha)) + k_sS + k_tT \tag{2.9}$$

where k_{sr} and k_{tr} are reflection and transmission constants (for the red band) that are used to weight the contributions S and T coming from the specular reflection and transmission directions. Similar equations are written out for the green and blue bands.

2.3.1.2 An Example Ray Trace

Fig. 2.4 illustrates an example scene in which a ray through a pixel is being traced. In this scene, for each surface i, I_i is the intersection point, R_i is the reflected ray, T_i is the transmitted ray and N_i is the unit normal to the surface at the intersection point.

A ray that is spawned from the eye through the pixel hits surface 1 at I_1. This point gets illumination from 2 spawned rays, R_1 and T_1 along the reflection and transmission directions. T_1 does not hit anything, but R_1 hits surface 2 at I_2, and it gets illumination from T_2 and R_2. Lastly R_2 hits surface 3 at I_3 and gets its illumination from R_3 and T_3. Now we apply a shading

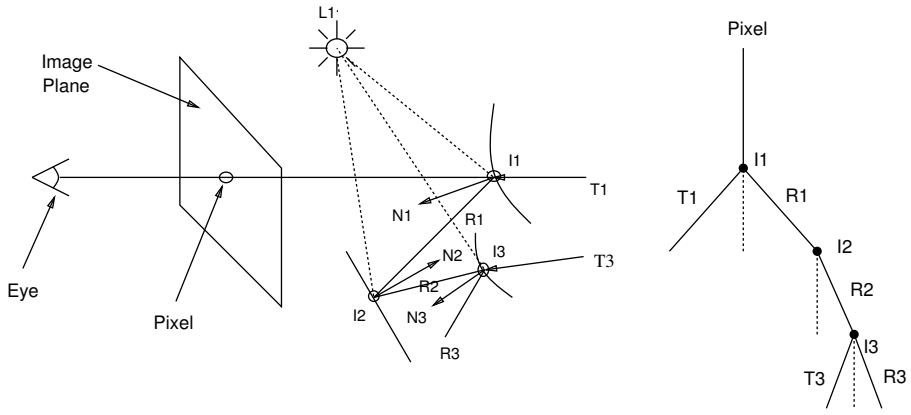


Figure 2.4: A Sample Ray Trace

model at I_3, I_2 and I_1 successively. We also need to see if any of these points is in shadow. For this a ray is spawned from each intersection point to each (point) light source as shown in the figure. Points in shadow will not be visible from that particular light source and hence will not have any direct specular or diffuse contribution due to it. The intensity computed at I_3 (with contributions from R_3, T_3 and the diffuse component if I_3 is not in shadow) will become the intensity of R_2. The intensity of I_2 will be computed in the same way and will become the intensity of the reflected ray R_1. Finally the intensity computed at I_1 will be the intensity seen at the eye.

This is the simplest form of ray tracing in computer graphics used for synthesizing images. The question that needs to be answered over and over again in this technique is 'Given a ray and a scene of objects, what is the closest object (if any) hit by the ray?'. For each pixel, we end up with a tree of rays.

Let us now look at an example to understand the amount of computation required to render a moderately complex scene using ray tracing.

2.3.1.3 Computation Expense: An example

Consider a scene containing a thousand spheres that need to be rendered. Let two light sources illuminate the scene. Typical workstations have a display resolution of 1000×1000 pixels. Assuming one ray is spawned from the eye point for each pixel, we have one million primary rays. Let us assume that on the average there are 4 additional rays spawned for each primary ray, accounting for reflection and refraction rays, and rays to light sources. Thus there are totally five million rays that need to be traced in an environment containing a thousand object primitives. To get an idea of how many operations it takes to compute a ray-sphere intersection, let us illustrate the procedure [16].

2.3.1.4 Ray-Sphere Intersection

Define a ray as:

$$R(t) = R_o + \vec{R}_d t, \qquad t > 0$$

where

$$\begin{aligned} R_{origin} \quad & \equiv R_o = [X_o, Y_o, Z_o] \\ R_{direction} \quad & \equiv \vec{R}_d = [X_d, Y_d, Z_d] \\ \text{where} \quad & X_d^2 + Y_d^2 + Z_d^2 = 1 \end{aligned}$$

Define a Sphere as:

$$\begin{aligned} &\text{Center} \equiv S_c = [X_c, Y_c, Z_c] \\ &\text{Radius} \equiv S_r \end{aligned}$$

Sphere's surface is the set of points $[X_s, Y_s, Z_s]$ where

$$(X_s - X_c)^2 + (Y_s - Y_c)^2 + (Z_s - Z_c)^2 = S_r^2$$

To solve for the intersection, substitute the ray equation into the sphere equation. Solve for t.

In ray parameter space,

$$X = X_o + X_d t$$
$$Y = Y_o + Y_d t$$
$$Z = Z_o + Z_d t$$

Substitution in the sphere equation results in

$$(X_o + X_d t - X_c)^2 + (Y_o + Y_d t - Y_c)^2 + (Z_o + Z_d t - Z_c)^2 = S_r^2$$

In terms of t, this is

$At^2 + Bt + C = 0$, where

$$A = X_d^2 + Y_d^2 + Z_d^2 = 1$$
$$B = 2 * \{X_d(X_o - X_c) + Y_d(Y_o - Y_c) + Z_d(Z_o - Z_c)\}$$
$$C = (X_o - X_c)^2 + (Y_o - Y_c)^2 + (Z_o - Z_c)^2 - S_r^2$$

The quadratic has the solution

$$t_0 = (-B - \sqrt{B^2 - 4C})/2$$
$$t_1 = (-B + \sqrt{B^2 - 4C}/2$$

If the discriminant is negative, the ray misses the sphere. Else, the smaller, positive root is the closer intersection.

If an intersection is found, the intersection point is calculated as follows:

Calculation	Multiplies	Adds/Subtr.	Compares	Divides	sq.root
A,B,C	7	8	0	0	0
Discriminant	2	1	1	0	0
t_0	1	1	1	0	1
t_1	1	1	1	0	0
Inters. point	3	3	0	0	0
Normal	3	3	0	0	0

Table 2.1: Operations in Ray-Sphere Intersection.

$$\vec{r}_{intersect}(x_i, y_i, z_i) = [X_o + X_d t, Y_o + Y_d t, Z_o + Z_d t]$$

and the unit normal is given by

$$\vec{r}_{normal} = [(x_i - X_c)/S_r, (y_i - Y_c)/S_r, (z_i - Z_c)/S_r]$$

The operation count for each step in the intersection calculation is illustrated in Table 2.1. In the worst case, there is a total of 17 adds/subtracts, 17 multiplies, 1 square root and 3 compares. Let us normalize all these to adds or subtracts. Adds, subtracts and compares cost nearly the same. Most workstations today contain floating point hardware. This usually makes a multiply nearly the same cost as an add. A square root takes the cost of about 12 adds. The total cost is now (17+17+12+3) 49 adds or say, 49 operations.

In the absence of any search structure, the total cost of ray tracing this environment is given by

Total Cost (in operations)= total rays * total objects * total intersections

$$= 5,000,000 * 1000 * 49$$

$$= 245,000,000,000$$

If we assume that a floating point operation takes about 1μsec., the total time

for the ray tracing is given by

$$\begin{aligned} \text{Total time} &= \frac{245,000,000,000 * (1 * 10^{-6})}{60 * 60} \text{ Hours} \\ &\approx 68 \text{ Hours!} \end{aligned}$$

At 30 frames per second, it would take about 2041 hours or 85 days to make 1 second of an animation sequence involving this scene!

2.3.2 Distributed Ray Tracing

Section 2.3.1 listed the assumptions made by the traditional ray tracing algorithm in approximating the integral in Equation 2.1. In this section we will describe a generalization of the standard ray tracing algorithm called *distributed ray tracing* [10][9] for enhanced photo-realism. Our interest in distributed ray tracing is twofold: it demonstrates how traditional ray tracing can be extended to simulate a variety of lighting effects that occur in real life environments (thus obtaining a more accurate estimate of Equation 2.1), at the expense of additional rays; and secondly, the computational expense involved further justifies the use of search structures for performance improvement. Distributed ray tracing relaxes the assumptions of standard ray tracing by spawning multiple rays per pixel according to some probability distribution. The exact distribution used and the origin and direction of the spawned rays depends on the effect that is being simulated. Let us examine some of them.

2.3.2.1 Gloss

The reflection integral in Equation 2.1 is approximated by a δ function. For specular surfaces, at each intersection point, exactly one reflection ray is

spawned along the mirror reflection direction. In real life, reflections are hazy, since most surfaces are not purely specular, due to surface defects, for instance. The distinctness with which a surface reflects light is called *gloss*. Gloss can be computed by distributing secondary rays about the mirror reflection direction. The ray contributions are weighted, with directions closer to the mirror direction contributing more to the total intensity. This intensity replaces the specular component in Whitted's lighting model.

2.3.2.2 Translucency

If there are objects in the environment that transmit light, then the reflection function in Equation 2.1 is replaced by the transmittance function T, and the integral is evaluated over the hemisphere behind the surface.

T is usually approximated by a δ function. Translucency is a characteristic that will result in blurred images of objects seen through transparent objects. To achieve translucency effects, transmission rays are distributed around the transmittance direction (defined by the refractive index of the surface material and the angle of the incoming ray), just as reflection rays are distributed around the mirror reflection direction.

2.3.2.3 Penumbras

The naive ray tracing algorithm assumes all light sources in the environment are point sources, resulting in sharp shadows. When light sources are of finite size, which is more common, the shadows are soft. Soft shadows occur because points in the scene might be partially obscured from the light source

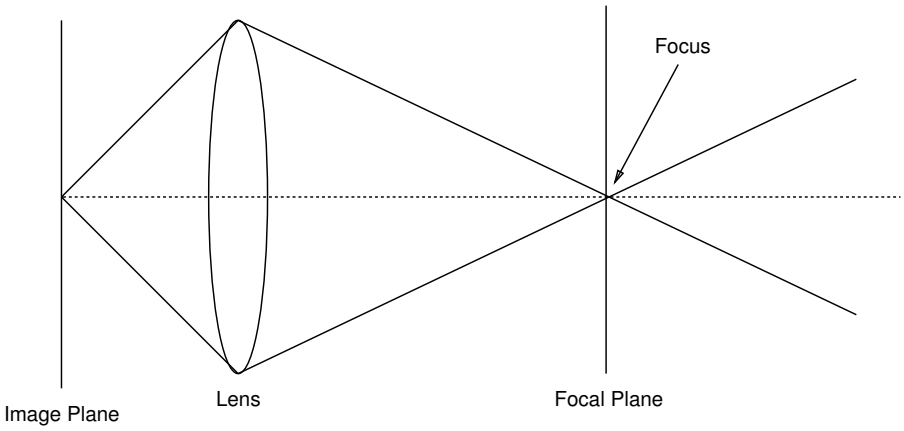


Figure 2.5: Lens Geometry

by intervening objects. The reflected intensity due to such a light is proportional to the solid angle of the visible portion of the light [11].

Shadows are normally calculated by spawning one ray from each intersection point to each light source. Since light sources are points in the standard ray tracing method, this is adequate. When light sources are of finite size, this is not sufficient, since different parts of the light source contribute to the illumination of each surface point. Penumbras can be calculated by spawning a collection of rays from each intersection point to each light source. The distribution of the shadow rays must be weighted depending on the projected area and the brightness of the different parts of the light source. The number of shadow rays traced to a region of the light source should be proportional to the amount of light energy coming from that region.

2.3.2.4 Depth of Field

Both cameras and our eyes have a finite lens aperture, and hence, a finite depth of field. Computer graphics has been based on the pinhole camera

model, with all objects in sharp focus. The focal point on a lens is the point where all rays converge, after refraction through the lens. The plane containing the focal point and orthogonal to the line of sight is the focal plane. Fig. 2.5 illustrates this.

Depth of field occurs because of the finite lens size. Visible surfaces and hence, shading could be different from different parts of the lens. These are not accounted for in the pinhole camera model where each point in the focal plane is looked at from a single point.

To account for depth of field effects, visibility calculations from parts other than the center of the lens must also be accounted for. For this, start with the ray from the center of the lens and determine the location on the image plane through which the primary ray is to be traced, just as in the standard algorithm. The focal point is located on this ray so that its distance from the eye point is the focal distance of the lens. Let this point on the image plane be p. A point on the lens is obtained by jittering a location selected from a prototype pattern of lens locations. The primary ray starts at this location and passes through the focal point. This ray is traced using the standard ray tracing algorithm. This procedure essentially samples the lens and objects not in focus are rendered properly.

2.4 Extensions to Ray Tracing

In addition to distributed ray tracing, there are other important extensions to ray tracing. The rendering equation proposed by Kajiya [24] generalizes a variety of known rendering algorithms. Approximations to this equation give rise to rendering techniques such as the standard scanline algorithms with hidden surfaces removed, ray tracing, distributed ray tracing and the radiosity

methods. Our main interest in the rendering equation is that it can be solved using ray tracing. Kajiya points out a variety of hierarchical sampling (or equivalently, variance reduction) techniques to solve the rendering equation. Specifically, an enhanced version of stochastic ray tracing, using techniques called importance sampling and path tracing, this has resulted in modeling environments which contain surfaces with a variety of surface characteristics.

Another important rendering technique for modeling global diffuse illumination is the radiosity method [19][8][50][7]. Ray tracing approaches to radiosity methods have also been proposed [51][52][4].

2.5 Conclusions

We have outlined the basics of ray tracing and some of its extensions for greater photo-realism. Although the more advanced forms of ray tracing such as distributed ray tracing are more expensive due to the larger number of spawned rays, the computation and complexity involved for tracing each ray remains the same. If we can trace each ray a little faster, then the overall performance will improve regardless of the effects we are trying to simulate. In the following chapters, our goal will be design data structures that are able to take advantage of scene properties to reduce the cost of tracing each ray.

Chapter 3

SEARCH STRUCTURES IN RAY TRACING

Having described the important aspects of ray tracing, we now turn to the need for accelerating ray tracing through the use of search structures. We will look at some of the important search structures currently being used.

3.1 Need for Accelerating Ray Tracing

In tracing each ray, we have to determine its closest intersection (if one exists) with an object in the environment. The simplest strategy is to test the ray for intersection with all the individual object primitives in the environment. The total cost of rendering becomes

$$C_{trace} = n_{rays} * n_{objects} * C_{pr}$$

where

C_{trace} : total cost of the rendering in terms of object intersections.

n_{rays} : total number of rays spawned.

$n_{objects}$: total number of objects.

C_{pr} : the average cost of intersecting an object primitive.

In this equation, there are three terms that contribute to the total cost; the total number of rays spawned, the total number of objects and the average object

intersection cost. The number of spawned rays is dependent on the image resolution, characteristics of the surfaces in the environment (highly specular surfaces would cause a large number of higher generation rays to be spawned), and the different lighting effects that we are trying to simulate. For instance, to generate penumbras using distributed ray tracing, several rays need to be spawned from each intersection point to each light source.

The total number of ray-object intersections is dependent on the total number of objects in the environment. In the absence of any search structure, all of them need to be examined for each spawned ray. Our goal will be to use search structures to reduce the total ray-object intersections. Reducing this parameter usually has the greatest advantage in performance improvement.

The cost of computing an intersection between a ray and an object primitive depends on its geometry. While this computation is inexpensive for object primitives such as polygons and spheres, this is not the case for higher order surfaces such as bicubic patches, often taking up hundreds of floating point operations. Techniques to reduce this cost include converting complex geometric objects to a simpler representation. For instance, subdividing a bicubic surface until each region can be approximated by a planar polygon [27] is one method to simplify the intersection calculations. Using simple volumes in place of object primitives is another alternative, when this substitution is acceptable [48].

Our focus will be directed towards reducing the ray-object intersections through the use of geometric search structures. A search structure will help us examine only a fraction of the environment at a small cost; that of traversing it. Most often, traversing a search structure involves inexpensive plane intersections or bounding volume tests. The advantage of these search

structures is especially significant in complex environments containing tens of thousands of object primitives.

The two main methods of reducing the total number of ray-object intersections is through the use of space partitioning hierarchical structures and those based on bounding volumes.

3.2 Space Partitioning Structures

In space-partitioning structures, 3-dimensional space is divided into a collection of convex regions by partitioning planes, usually axis-aligned. Space is recursively partitioned until each region contains a small number of object primitives. The partitioning planes help determine an order of regions along the path of any ray of arbitrary origin and direction. Only the objects in these regions need to be examined for a possible intersection. Processing stops once an intersection is found or there are no more regions to examine.

Let us look at some of the common space partitioning structures being used in ray tracing.

3.2.1 Uniform Subdivision

In this method, the three-dimensional extents of the scene represented by an axis-aligned parallelepiped are subdivided uniformly along all three dimensions, resulting in a grid of equal sized 'voxels' (abbreviation for volume elements, each represented usually as a rectangular parallelepiped). The most common examples of this are the ARTS method [15] and more recently, the work of Cleary [6].

Since the subdivision creates equal sized voxels, tracing a ray through a grid is very similar to the 2D problem of drawing a line on a raster grid.

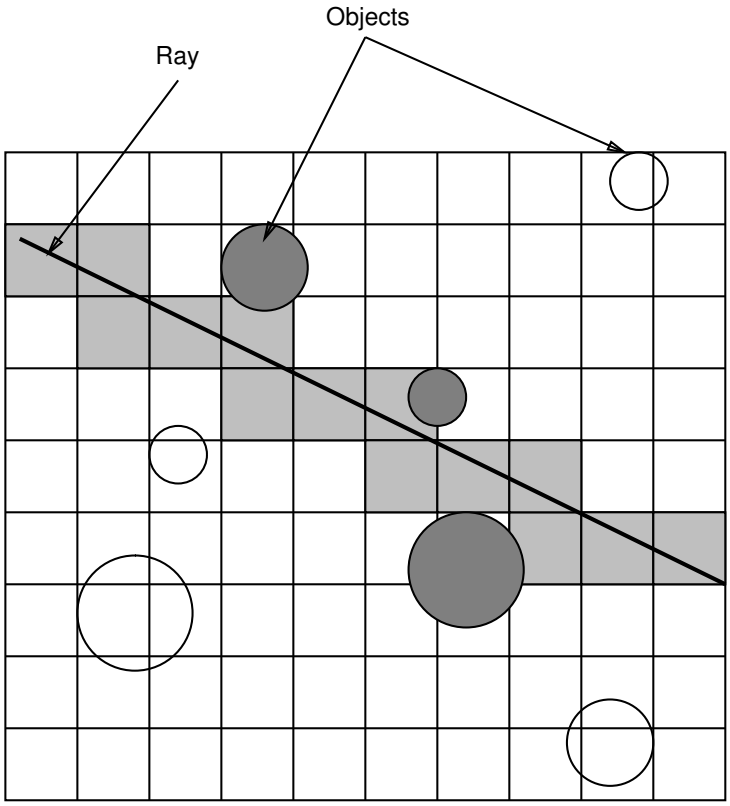


Figure 3.1: Uniform Subdivision (2D example).

The uniform size of the voxels permits the use of incremental techniques in identifying the cells visited by any ray. One difference from the line drawing algorithm is that whereas in 2D it is sufficient to identify cells that are close to the actual line being drawn, tracing rays through a 3D grid requires all cells visited by the ray be identified. This is because we are searching for the closest intersection with an object in the environment, and thus, all the cells must be visited in order. Once an intersection is determined, the search can be terminated (and the ray discarded) since no other intersection can be closer than the one already determined.

Fig. 3.1 illustrates uniform subdivision in 2D. The shaded cells and objects are examined by the ray in the search for the closest intersection. Al-

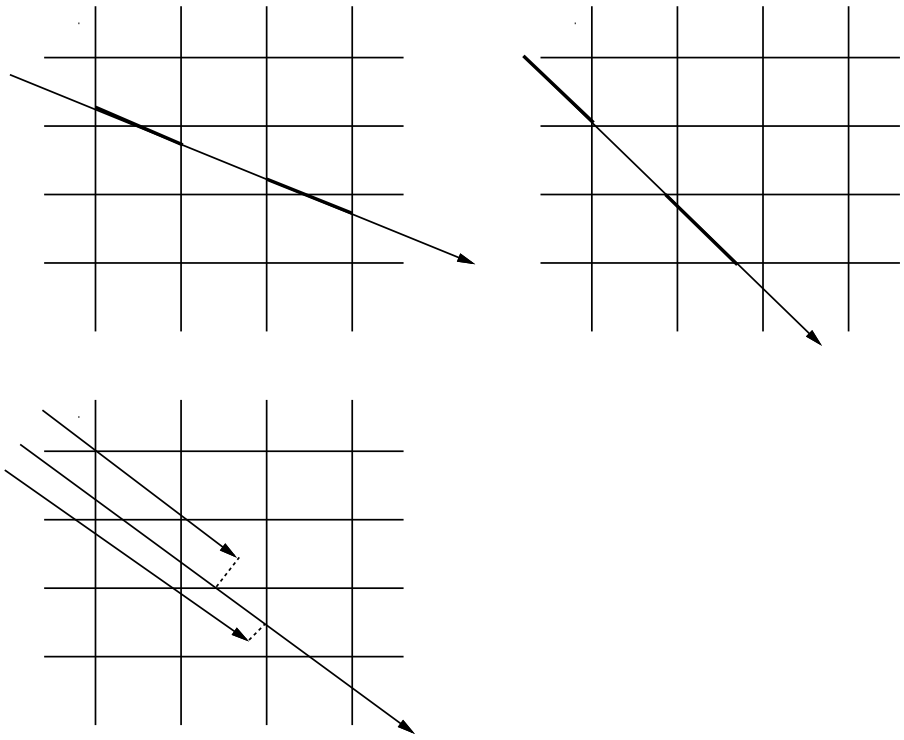


Figure 3.2: Geometry for Next Cell Calculations.

though the resolution of the grid can be increased to reduce the number of objects examined, this increases the expense of traversing empty regions of space and also increases the storage requirements very quickly.

The strategy used to move from cell to cell is the part that is critical to the performance of this technique. As our implementation follows the method described in [6], we will describe this method of cell traversal next.

3.2.1.1 Cell Traversal

Fig. 3.2 shows the geometry used in the next cell calculations (a 2D example). The ray in Fig. 3.2 enters a new cell either by passing from top to bottom through a horizontal wall, or by passing from left to right through a vertical

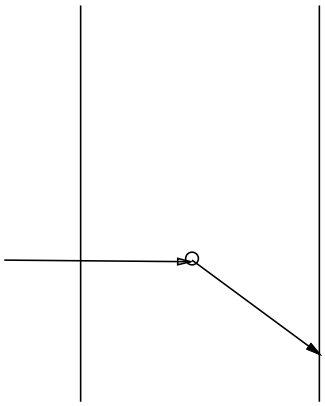


Figure 3.3: Initalization Calculations.

wall. The distance along the ray between vertical walls is a constant, δx, and there is a similar distance, δy, for the distance between horizontal walls. In 3D, there is also δz. The problem is to determine what type of crossing will occur next; whether a horizontal or vertical wall will be encountered. This is the role of dx and dy, the total distance the ray has traveled from some origin to the next crossing of a vertical or horizontal wall respectively. If $dx < dy$, the next crossing will be a vertical wall and the next cell, a horizontal neighbor. If $dy < dx$, a horizontal wall will be encountered and the next cell will be a vertical neighbor. dx is updated by adding δx when a vertical wall is crossed. dy is updated by adding δy when a horizontal wall is crossed. The cell address is maintained by two integers i and j. The direction flags of the ray are stored by px and py.

For all of this to work, the values of $dx, dy, dz, \delta x, \delta y, \delta z$ must all be properly initialized. The ray is assumed to be inside the grid. This is very much true when a ray has been reflected (or transmitted) from an object in the environment. To adapt this to first generation rays (the view point could be outside the grid), the point where the ray enters the grid needs to be computed.

Secondly, all the coordinates must be scaled so that each voxel of the grid is a $1 \times 1 \times 1$ cube. The scaled grid will be used for the 'next cell' calculations. Once the new cell has been identified, it can be mapped back to world space to identify the next cell that is visited by the ray.

We next show how to initialize all the quantities being used in the traversal calculations. The ray origin is assumed to be at (x, y, z). From Fig. 3.3,

$$\begin{aligned} dx &= (\lfloor x \rfloor + 1 - x)\ \delta x, \qquad \text{if ray is going towards right,} \\ &= (x - \lfloor x \rfloor)\ \delta x, \qquad \text{if ray is going towards left} \end{aligned}$$

where $\lfloor x \rfloor$ is the greatest integer less than x. A similar procedure is followed for dy and dz. To calculate δx, we need the direction cosines cx, cy and cz. From Fig. 3.3,

$$\delta x = \sqrt{cx^2 + cy^2}/cx$$

This is easily extended for three dimensions. δy and δz are similarly calculated.

This results in the 2D cell traversal algorithm described in Algorithm 3.1.

Indexing the cell array is expensive. In 3D, another index, k is needed in addition to i and j. It is better to maintain a linear array with an index p.

$$p = i * n^2 + j * n + k$$

The multiplies can be avoided by maintaining p directly with appropriate values for px, py and pz. Each time i is incremented, p is incremented by $\pm n^2$; each time j is incremented, p is incremented by $\pm n$.

```
Initialize px, py, δx, δy, dx, dy, i and j.
repeat
        if (dx ≤ dy)
        {
          i = i + px
          dx = dx + δx
        }
        else if (dy ≤ dx)
        {
          j = j + py
          dy = dy + δy
        }
until an intersection is found in cell(i, j)
```

Algorithm 3.1: Cell Traversal (2D).

A hash table is used to store the cells for efficient utilization of array space. The hashing index is simply p mod M, where M is the size of the table. p can be checked at the end of the loop to see if it exceeds M.

Finally, each ray has to be terminated properly. This is done by checking to see if the ray has exited the 3D volume. Recall that dx, dy and dz keep track of the total distance of the ray from some origin. The distances at which the ray exits the volume (along each of the three dimensions) is determined by intersecting the ray against the faces of the volume. Let these distances be sx, sy and sz. To start with, $sx = (n - x)\delta x$, assume n is the resolution of the grid along the X axis. sy and sz are similarly calculated.

The cell traversal algorithm in 3D, including termination and hashing, is illustrated in Algorithm 3.2.

3.2.2 BSP Trees

A binary space partitioning (BSP) tree is any binary tree structure used to recursively partition space. BSP trees have been used to determine vis-

```
Initialize px, py, pz, δx, δy, δz, dx, dy, dz, sx, sy, sz and p.
repeat
    if (dx ≤ dy) && (dx ≤ dz)
    {
      if (dx > sx) exit;
      p = p + px
      dx = dx + δx
    }
    else if (dy ≤ dx) && (dy ≤ dz)
    {
      if (dy > sy) exit;
      p = p + py
      dy = dy + δy
    }
    else
    {
      if (dz > sz) exit;
      p = p + pz
      dz = dz + δz
    }
    if (p > M)p = p − M
until an intersection is found in cell with hash key p
```

Algorithm 3.2: Cell Traversal (3D).

ible surfaces [46][13][35][40][14][34], polyhedral set operations [47] and shadow generation [5].

The earliest use of BSP trees was by Shumacker [42]. Shumacker's algorithm partitions the environment into a set of clusters using hand-picked partitioning planes. For this, the objects in the environment must be *linearly separable,* i.e., there must exist a plane which partitions the objects into two nonempty sets without intersecting any of the objects. With the environment divided into two subsets, each of these sets is recursively subdivided until each *cluster* contains a small number of objects. An example is shown in Fig. 3.4. 1, 2 and 3 are object clusters. A, B, C and D are the partitions created by the two planes, indicated by the bold lines. The binary tree that is built contains

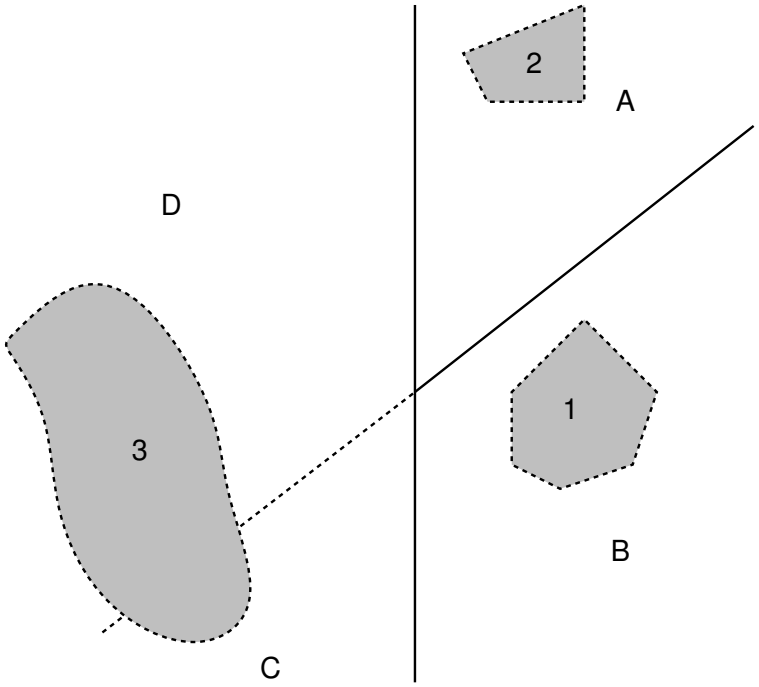


Figure 3.4: Determining Cluster Priority.

partitioning planes in its interior nodes and clusters of objects at the leaf nodes.

Shumacker's method involves hand-picking the partitioning planes. The technique of Fuchs and Naylor [13][14] automates the process of choosing the planes. This is very important for complex environments. Their method as originally described, is, however, restricted to polygonal scenes, since the partitioning planes are the planes of the polygons that determine the scene.

In Fuchs and Naylor's BSP tree, the construction of the tree begins by choosing a partitioning plane that contains a polygon from the input scene. The remaining polygons in the scene are partitioned into three sets, those on the partitioning plane, and those that are on the two sides of the plane. Polygons which cross the partitioning plane are split and each piece placed in the appropriate set. Thus the environment need not be linearly separable. As in the earlier algorithm, the two subsets are recursively subdivided, picking at

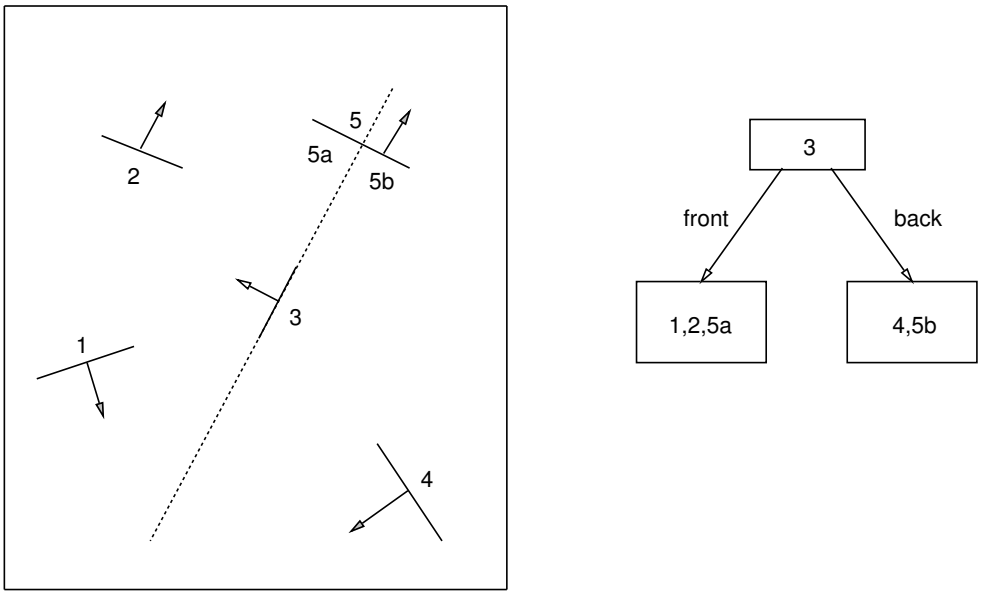


Figure 3.5: Determining Visible Surfaces Using BSP Tree.

each step one of the polygons as the partitioning plane. The process ends when no polygons remain. The BSP tree thus built now contains a polygon at each node with the left and right subtrees containing the polygons on either side of the plane of the polygon.

An example is shown in Fig. 3.5. Here the plane at the root of the hierarchy is the plane containing polygon 3. This intersects polygon 5, which is split into 5a and 5b. Polygons 1, 2 and 5a are considered to be in front of the root plane while 4 and 5b are in the back of the root plane. The direction of the plane normal vector distinguishes the front side from the back. The BSP tree after partitioning by plane 3 is also shown in Fig. 3.5. The process is recursively applied to the two children of the root node.

Given a view point, the constructed BSP tree can be traversed in a back to front or front to back order. For visible surface generation, a back to front traversal of the polygons is used, since polygons in front of a parti-

tioning plane (and closer to the view point) cannot be blocked by any of the polygons on the farther side of the partitioning plane. The necessary ordering can be determined with the help of the partitioning planes once a view point is specified.

For instance, consider Fig. 3.5. Let us assume the viewpoint is on the back side (the side containing the plane normal is the front side and the other side is the back side) of polygons 4, 3 and 5b. The partitioning plane at the root of the BSP tree is the plane of polygon 3. Since the viewpoint is on the back side of 3. polygons on the front side of 3 (polygons 1, 2 and 5a) are processed before those on the back side of 3. In the BSP tree in Fig. 3.5, the left node polygons is processed, then the root polygon and finally, the polygons on the right child node. The partitioning planes stored in the two child nodes determine the order in which their respective polygons are processed. The relation of the position of the viewpoint with respect to the partitioning plane determines this order.

As long as the objects in the scene do not move, the back to front ordering can be reconstructed for any view point.

Kaplan [25] implemented a special case of the BSP tree for ray tracing. In this implementation, axis-aligned planes are used to partition space. At each step of the subdivision, three partitioning planes are used to divide space into eight equal sized octants. Each object in the scene is tested against each octant to see if any part of its surface intersects it. If it does, then it is added to the list of objects for that node. At the end of the first step of the subdivision, we have eight lists of objects, one for each of the eight nodes. At the root, the partitioning is by a plane orthogonal to the X axis, at the next level by two planes orthogonal to the Y axis and at the last level, by four planes orthogonal

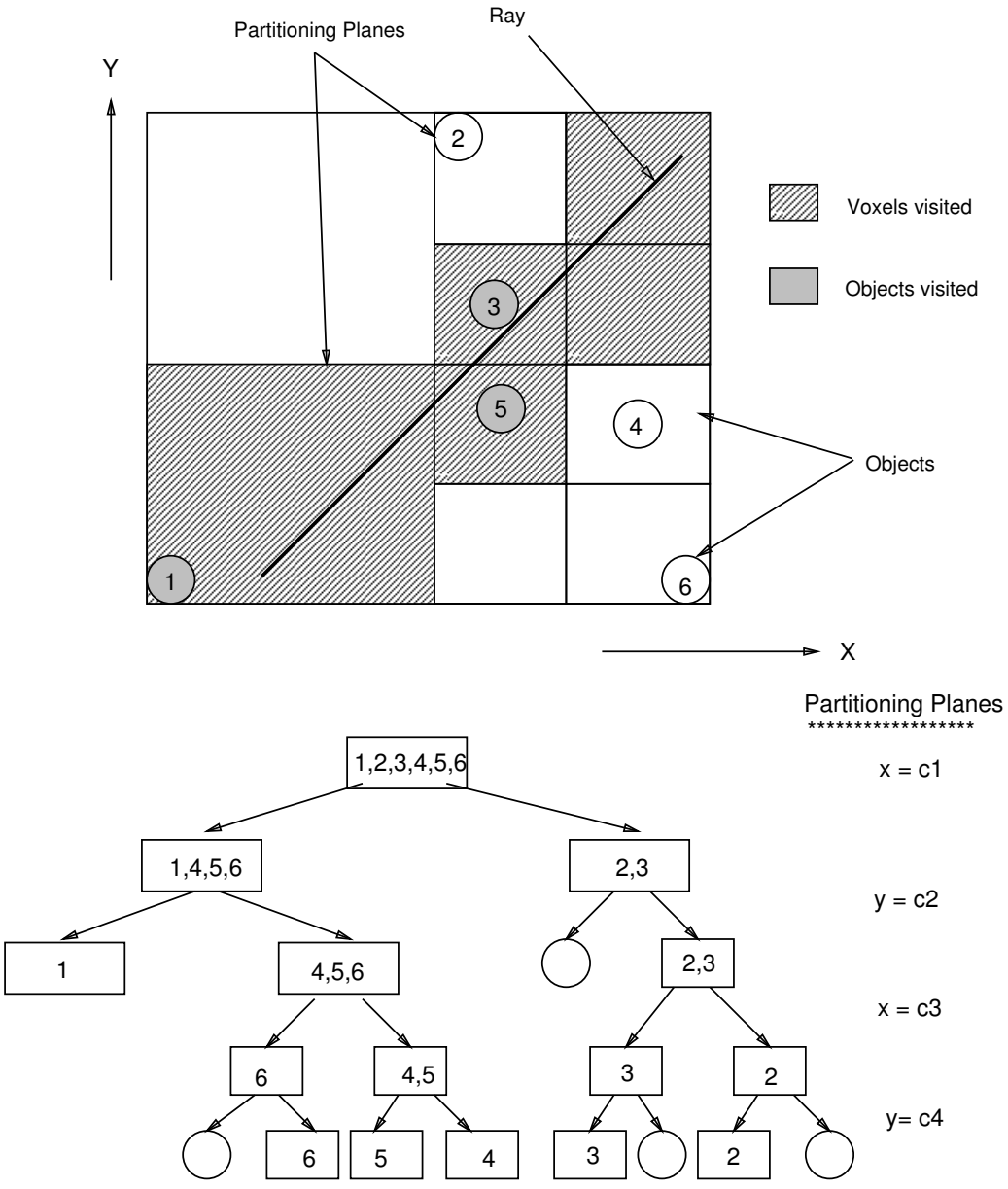


Figure 3.6: Kaplan-BSP Tree.

```
Input: obj_list, bsp_root, ray.
Output: inters
kaplan_traverse (object_list, bsp_root, scene_bounds)
{
  while ( !inters.found && (ray.origin within scene_bounds) )
  {
      r = bsp_root
      while (r == INTERNAL_NODE)
      {
          dim = r.plane_dim
          if (ray.origin[dim] < r.plane_val
              r = r→left
          else
              r = r→right
      }
      "Test objects at LEAF node r for an intersection, update inters.found, inters.point"
      If (!inters.found)
      {
          "Determine point through which ray exits LEAF NODE, record exit point",
          "Extend exit point along ray direction into the next region"
          "compute new ray origin"
      }
  }
  return (inters)
}
```

Algorithm 3.3: BSP (Kaplan) Tree Traversal.

to Z axis. The recursive subdivision continues until the leaf nodes of the tree contain either a small number of primitives or their size becomes smaller than a set threshold. At the end of the subdivision, all the leaf nodes contain lists of object primitives. Note that the subdivision adapts itself to the scene structure since only regions that contain object primitives are subdivided. A 2D example of a Kaplan-BSP subdivision and the corresponding tree are shown in Fig. 3.6.

To determine a ray-object intersection from the BSP tree, the origin of the ray is compared against the partitioning planes in the tree, starting from the root of the tree. The leaf node containing this point can be determined. All

primitives contained at this node are intersected with the ray. If an intersection is found internal to this region, the closest of these is the required intersection. Otherwise, the face through which the ray exits this region is determined by performing an intersection between the ray and the six faces of the region. The ray is then extended a small amount, depending on the size of the smallest region in the tree. This is done as follows (the following procedure also applies to the octree).

During the construction of the BSP tree, the length of the smallest side of any of the voxels, say l_{min}, is recorded. The next voxel visited by the ray is found by moving perpendicular to the face through which the ray exits the current voxel. If the movement is limited to be less than l_{min} (for example, $l_{min}/2$), then the next voxel will not be missed. If the exit point is on an edge, it is necessary to move perpendicular to both faces sharing that edge, and in three directions if it is on a corner.

Now the new point is contained in the region that is visited next by the ray. This region is determined in a similar fashion as for the BSP tree, and search for the closest intersection continues.

The traversal method is illustrated in Algorithm 3.3.

3.2.3 Octree

The octree hierarchy used by Glassner [17] performs a subdivision identical to Kaplan's BSP tree. Each level of the octree corresponds to three levels of the BSP tree. The difference lies in the way Glassner stores the octree. While Kaplan builds a binary tree, Glassner uses a combination of a hash table and linked lists, which results in considerable savings in pointer space. Each node in the hierarchy has a uniquely defined name. This name

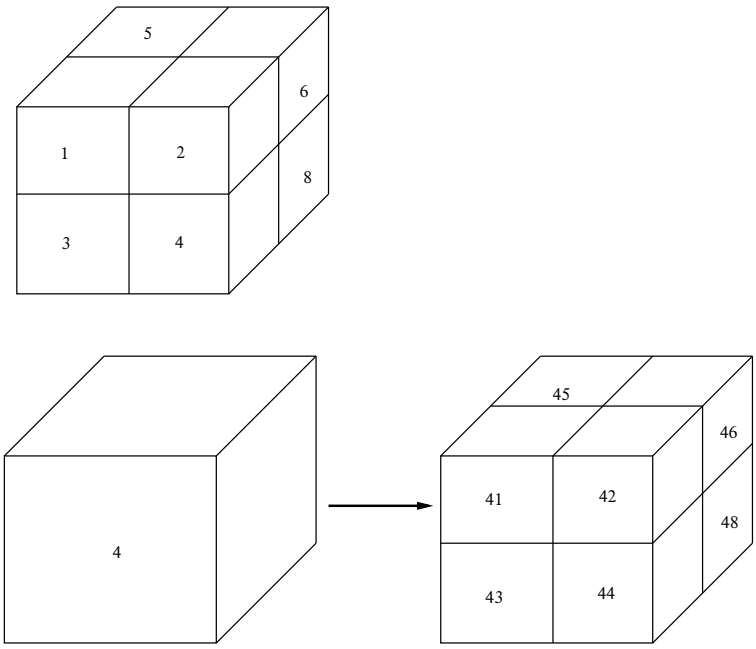


Figure 3.7: Octree Naming Convention.

is constructed using the convention of labeling the children of a node by the sequence of numbers 1 through 8. When a node is subdivided, its children derive their names by appending their single digit (identifier) to their parent's name. See Fig. 3.7. Given the name of a node, the name of its child is obtained by multiplying it by 10 and adding the appropriate digit. To access data associated with a node name (for example, accessing an object list), the name is used to retrieve a pointer from the hash table. Glassner computed the name modulo the size of the table as a hashing function. To retrieve an object list, determine which of the eight octants contains the point and consult the hash table for the status of that child. The child could have been further subdivided, or it might be a leaf node, in which case its object list can be accessed.

Traversing an octree is almost identical to the Kaplan-BSP tree. The main difference is in the method used to reach the leaf nodes of the octree.

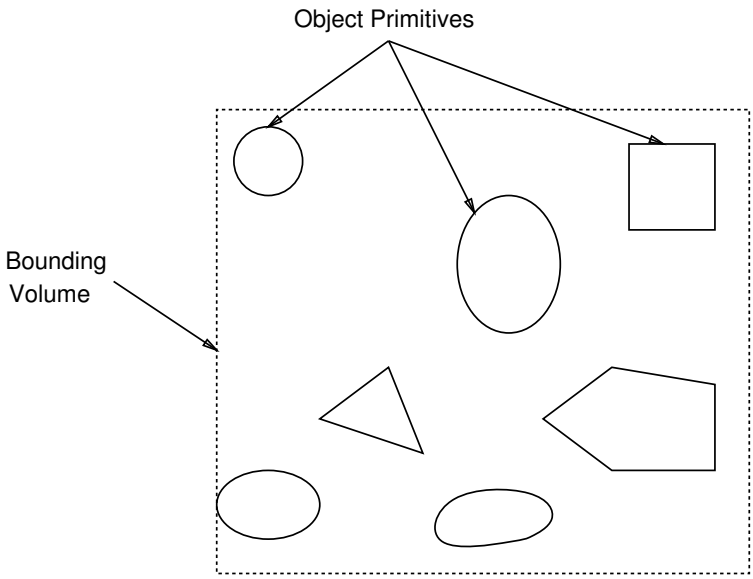


Figure 3.8: A Bounding Volume Enclosing Objects

While this is done in the octree by consulting a hash table and following link lists, the procedure in the Kaplan-BSP tree involves comparing the coordinates of the point (which is guaranteed to be inside the region of interest) against the partitioning planes in the tree.

Fig. 3.6 illustrates a 2D example of an octree. The search examines regions and objects along the ray path, with the regions examined shown by the shaded cells and objects. The subdivision is concentrated in the regions containing larger collections of objects.

3.3 Bounding Volume Hierarchies

Another widely used technique to reduce the ray-object intersections is through the use of bounding volumes. A bounding volume is a volume that surrounds one or more object primitives completely. Rays that miss a bounding volume also miss the object(s) contained by the bounding volume.

However, the geometry of a bounding volume must be simple so that it is not very expensive to test a bounding volume for intersection with a ray. At the same time, it should fit the objects inside as tightly as possible. A bounding volume enclosing object primitives is shown in Fig. 3.8.

Carrying this further, objects can be grouped together hierarchically [39][55] and bounding volumes constructed at each node of the hierarchy encompassing all the objects under that node. Thus, whole collections of objects can be pruned from further consideration of a ray by intersecting the bounding volume at any of the interior nodes of the hierarchy. This reduces not only ray-object intersections, but also avoids most of the ray-bounding volume intersections.

One important distinction from space partitioning structures is that bounding volumes are, in general, not disjoint. Space-partitioning structures create disjoint volumes, however, a partitioning plane can cross an arbitrary number of object primitives. In bounding volume structures, the regions enclosed by the bounding volumes can overlap but all the objects inside it are completely contained by the volume.

3.3.1 Automatic Bounding Volume Hierarchies

Some of the earliest bounding volume hierarchies required direct specification of the objects to be grouped together at each stage. While this is not unreasonable for small environments containing few primitives, it becomes inconvenient for very complex scenes. The particular strategy used to cluster the objects is critical to the performance of the hierarchy.

Goldsmith's [18] automatic bounding volume hierarchy (ABV) takes an important step in this direction. It provides a method to evaluate the cost

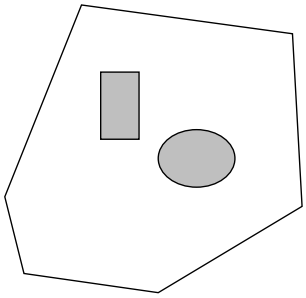


Figure 3.9: Hitting Probability.

of the hierarchy. For rays with an endpoint at a fixed distance from a bounding volume, the probability that a ray (whose directions are uniformly distributed) will hit the bounding volume is proportional to the solid angle subtended by the surface of the bounding volume. At large distances, for convex volumes like spheres and rectangular parallelepipeds, this is approximately proportional to the surface area [45].

We will prove this result by calculating the average projected area of the convex volume over all possible orientations. The probability that a ray will penetrate a volume is proportional to the area projected by the volume in a direction orthogonal to the ray direction.

Lemma 3.1 *The average projected area of a planar polygon over all possible orientations is one-half its surface area.*

Proof.

Consider a polygon P of area A and unit normal vector N. Let D be any projection direction. The projected area A_p of P orthogonal to D is given by

$$\begin{aligned} A_p &= A(N \cdot D) \\ &= A|N||D|cos\theta \\ &= Acos\theta \end{aligned}$$

where $\cdot$, represents dot product, θ is the angle between N and D, and $|N| = |D| = 1$. To get the average projected area of the polygon, average over the hemisphere of P.

$$\begin{aligned}
A_p(avg) &= \frac{1}{2\pi r^2}\int_0^{2\pi}\int_0^{\pi/2} A_p\,(rd\theta)\,(rsin\theta\,d\phi) \\
&= \frac{1}{2\pi r^2}\int_0^{2\pi}\int_0^{\pi/2} Acos\theta\,(rd\theta)\,(rsin\theta\,d\phi) \\
&= \frac{Ar^2}{2\pi r^2}\int_0^{2\pi}\int_0^{\pi/2} sin\theta\,cos\theta\,d\theta\,d\phi \\
&= \frac{A}{2\pi}\int_0^{2\pi}\int_0^{\pi/2} sin\theta\;\; d\,(sin\theta)\,d\phi \\
&= \frac{A}{2\pi}\int_0^{2\pi}\frac{1}{2}sin^2\theta\,\Big|_0^{\pi/2}\;\; d\phi \\
&= \frac{A}{2\pi}\int_0^{2\pi}\frac{1}{2}.\,d\phi \\
&= \frac{A}{2\pi}\int_0^{2\pi}\frac{1}{2}\,|_0^{2\pi} \\
&= \frac{A}{2\pi} * \pi \\
&= \frac{A}{2} \qquad\qquad \square
\end{aligned}$$

Theorem 3.1 *The average projected area of a convex surface over all possible orientations is one-quarter its surface area.*

Proof.

Any surface can be approximated arbitrarily closely by a polygonal mesh. If the surface is convex, projection along any direction D will be covered twice, once by the front facing polygons and once by the back facing polygons. The average projected area of the mesh is one-half the sum of the average projected areas of the polygons in the mesh. Using the result from Lemma 3.1 this approaches one-quarter the surface area of the convex surface as the number of polygons increases. $\square$

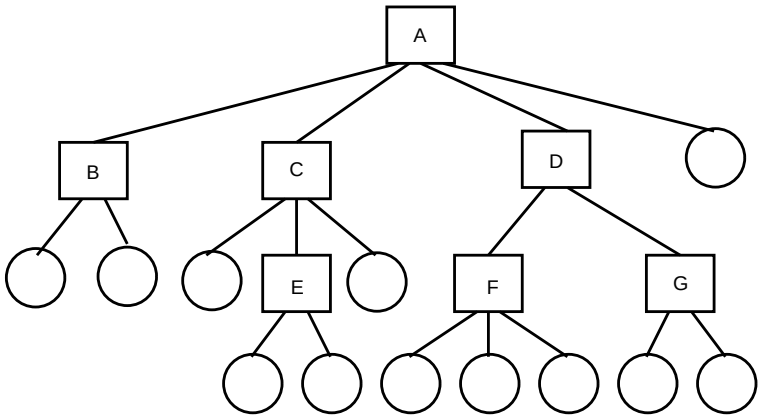


Figure 3.10: A Bounding Volume Hierarchy

In Fig. 3.9, A, B, C and D are convex regions. The probability that a ray will penetrate region B or C given that it penetrates A is given by $P(B|A)$ and $P(B|C)$. From the above theorem, a formula for obtaining these probabilities is given by

$$P(B|A) = \frac{Area(B)}{Area(A)} \tag{3.1}$$

$$P(C|A) = \frac{Area(C)}{Area(A)} \tag{3.2}$$

and so on. A simple hierarchy is shown in Fig. 3.10. Let us compute R, the expected number of bounding volume intersections for this hierarchy.

Assume the ray intersects the scene extent, i.e. intersects A.

#intersections at level $0 = 1$

#intersections at level $1 = 4P(A|A)$

#intersections at level $2 = 2P(B|A) + 3P(C|A) + 2P(D|A)$.

#intersections at level $3 = 2P(E|A) + 3P(F|A) + 2P(G|A)$.

and the expected number of intersections is given by

$$\begin{aligned} R &= 1 + 4P(A|A) + 2P(B|A) + 3P(C|A) + \\ &\quad 2P(D|A) + 2P(E|A) + 3P(F|A) + 2P(G|A). \end{aligned}$$

Using the surface area formula to compute the conditional probabilities, we can determine the expected number of intersections performed for each ray.

The hierarchy is built with a view to minimizing the total bounding volume surface area (in order to minimize the expected number of intersections). This is done by a heuristic tree search. Objects are inserted into the hierarchy, one at a time, and the tree is searched to find a suitable place for insertion. At any node, only the subtree that results in the smallest increase in the node's bounding volume area (when the object is inserted as a child of this node) is searched. If two or more children are found to have the same increase in bounding volume surface area, then all subtrees under these nodes have to be searched for a possible insertion point. At each level of the tree during the search, the new node is considered a prospective child of each node that will be searched. When the search reaches the leaf nodes, the new node and the leaf node are proposed as siblings of a new non-leaf node, replacing the old leaf node. Fig. 3.11 illustrates these two cases. After the search, the object is inserted in the tree where the increased cost of the hierarchy is minimized.

To traverse the bounding volume hierarchy, the ray is tested to see if it has an intersection with the bounding volume at the root node. If the ray penetrates the bounding volume, then all the node's children have to be examined. If not, its entire subtree can be removed from further consideration. This process continues recursively until there are no more nodes to be consid-

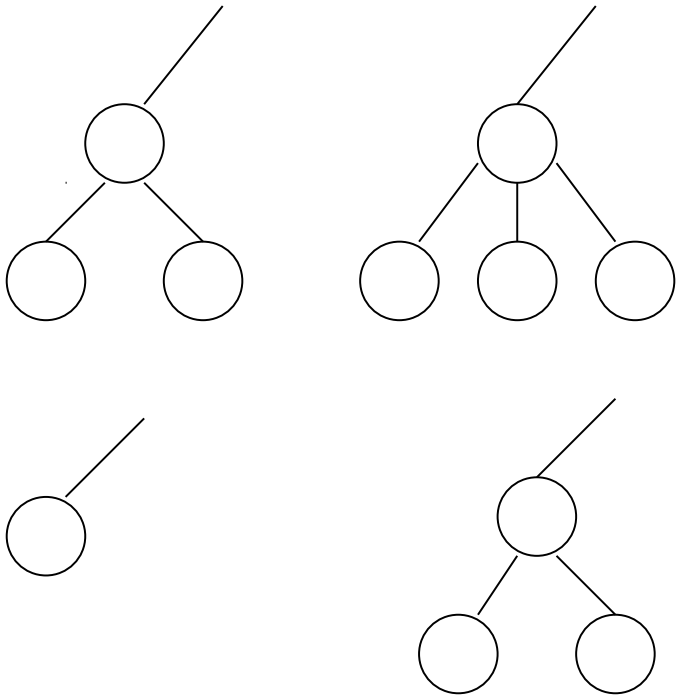


Figure 3.11: Inserting a Node into the ABV Hierarchy.

ered. Throughout the process, a record of the closest intersection, if any, is maintained and reported at the end of the traversal. The traversal algorithm is illustrated in Algorithm 3.4.

3.3.2 Using 'Tight' Bounding Volumes

We mentioned earlier that the geometry of the bounding volume must be kept simple in order to keep its intersection cost low. While this is true when the bounding volume encloses only a few object primitives, it might be possible to increase the complexity of bounding volumes when they contain large numbers of primitives, for example, at the root of a bounding volume hierarchy. A tighter-fitting bounding volume around large collections of object primitives could result in culling a large number of rays and eliminating them from further consideration.

```
Input: root of bounding volume hierarchy (bv_root), ray.
Output: Intersection (inters.pt - coordinates of intersection point,
inters.t - parameter value at the intersection.

bv_travers (bv_root, ray)
{
  if (root_bv == LEAF_NODE)
  {
    "Intersect with object at node, update inters.pt, inters.t"
  }
  else
  {
    for each child c of node
        bv_inters (root→child[c], ray)
  }
  return (inters)
}
```

Algorithm 3.4: ABV Hierarchy Traversal.

The best example of this is a method introduced by Kay and Kajiya [26]. In this approach, objects can be made to fit convex hulls arbitrarily tightly in exchange for a slower intersection computation. The bounding volumes used are many-sided parallelepipeds which are constructed by pairs of parallel planes. Each of these *plane-sets* is defined by a unit vector called the *plane-set normal*, and each plane in it is identified by its signed distance from the origin. Given a plane-set normal and an object primitive (bounded), there are two unique planes that bracket the object most closely. The region in space between these planes is called a *slab* and can be represented as a min-max interval associated with a plane-set normal, as shown in Fig. 3.12a.

A bounded region can be constructed as an intersection of several slabs appropriately oriented. An example is shown in Fig. 3.12b. In 2-space, two slabs are sufficient while in 3-space at least three slabs with linearly independent plane-set normals are required. Increasing the slabs makes the bound-

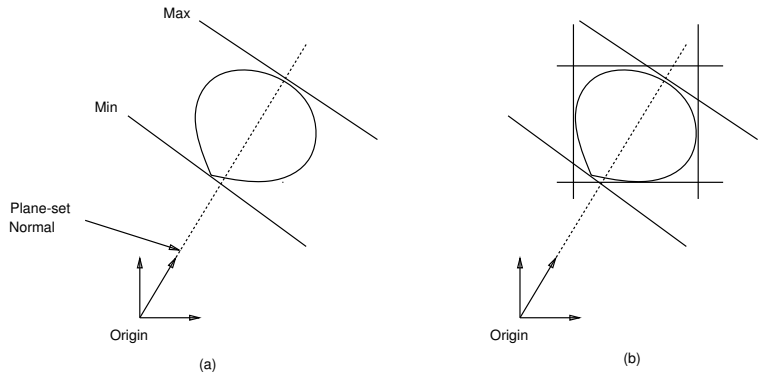


Figure 3.12: Slabs defining a Bounding Volume.

ing volume approximate the actual convex hull of the object(s) more closely, but at the same time increases the intersection cost. If there are s slabs in a bounding volume, a ray-bounding volume intersection test involves $2s$ ray-plane intersection tests. Each ray-slab intersection calculation defines an interval. If the intersection of all these intervals is empty, the ray misses the bounding volume. Otherwise, this interval gives the distance from the ray origin to the bounding volume intersection points.

If the same collection of plane-set normals is used for all objects (or collections of objects) in the environment, there are tremendous computational advantages in computing the bounding volume intersection.

Let the ray

$$R = \hat{a}t + b \tag{3.3}$$

intersect a slab to yield an interval along the ray. $\hat{a}$ is the ray direction and b, its origin. t is the ray parameter. We are interested in intersections for which $t > 0$. To determine this interval, we compute the points where R intersects the two planes that define the slab. The plane equation is given by

$$Ax + By + Cz - d = 0 \tag{3.4}$$

If $\hat{P}_i$ is the normal vector ($A,\ B,\ C$), then substituting Equation 3.4 into 3.3, in terms of ray parameter t, we get the solution

$$t = \frac{d_i - \hat{P}_i.b}{\hat{P}_i.a} \tag{3.5}$$

where d_i is the distance of the plane from the origin along the plane normal vector. From Equation 3.5, it is seen that each ray-slab intersection (which is two plane intersections) requires four dot products, two subtracts and two divides. This number is multiplied by the number of slabs making up the bounding volume.

However, if the plane-set normals are chosen in advance, all the dot products can be computed just once. Also, at this time, the reciprocal of the denominator of Equation 3.5 can be computed so that these divisions can be replaced by multiplications. Now each plane intersection is given by

$$t = (d_i - S)T \tag{3.6}$$

where $S = \hat{P}_i.b$ and $T = 1/\hat{P}_i.a$. This computation requires two subtracts, two multiplies and a compare for each slab contributing to the bounding volume.

Using a hierarchy is critical to performance. Constructing a bounding volume which bounds two or more parallelepipeds involves determining the minimum and maximum of all the plane constants associated with the plane-set normals of all the parallelepipeds.

Lastly, the hierarchy traversal algorithm processes objects in approximately the order in which they occur along the path of the ray. For this, the results of the bounding volume intersection (minimum and maximum distance from the ray origin) are used to keep a sorted list of objects (or bounding volumes containing a collection of objects in a hierarchy). The sorting of the objects is performed using a heap data structure.

Chapter 4

A COST MODEL FOR RAY TRACING HIERARCHIES

Most of the search structures developed over the last few years have relied on timing benchmarks to compare their performance against each other. While this is useful, it does not allow us to choose a search structure to use for a given input scene. What is more desirable is to have some idea of the performance of a search structure on a scene before the ray tracing is done. The performance of a search structure depends on the characteristics of the scene. It is possible that a structure that performs well on a particular scene might perform poorly against another. If this can be determined in advance, then a different structure may be substituted in its place for improved performance.

In this chapter we will develop a model for evaluating the cost of a space partitioning hierarchy, with a straightforward generalization to bounding volume hierarchies. This model will help us relate the computational costs of various techniques to appropriate parameters of the search structure. The statistical characteristics of the input scene are used in building the model. Our aim is to get an expression for the average number of intersection operations performed by each ray. In the next chapter, we will give two important application of the model, in predicting automatic termination criteria for ray tracing search structures, and in choosing a search structure for a given scene.

4.1 The Cost Model

Ray tracing hierarchies are built for the sole purpose of speeding up intersection searches for ray-object intersections. All of these structures help in drastically reducing the search space of each ray. This is accomplished in two different ways:

1. The search is ordered along the path of the ray, starting from its origin. This helps in terminating the search once an intersection is found.

2. The search examines only parts of the scene that are close to the ray. Even if no intersection is found, only a fraction of the scene would have been examined.

However, using a search structure introduces a new expense: the cost of traversing it. So long as the cost in traversing the structure is overwhelmed by the gains in reducing the ray search space, we are improving performance. The question is, what is the tradeoff?

4.1.1 Search Structure Costs

We can identify two major costs involved in using a search structure:

1. The cost in examining the scene, $C_{sc}(h, s)$(h is the height and s is a search structure). This is the cost of performing ray-object intersections and ray-bounding volume intersection tests (when object primitives are enclosed by bounding volumes).

2. The cost in traversing the search structure, $C_{tr}(h, s)$. This is the cost of going down the hierarchy to the leaf nodes. Depending on the ac-

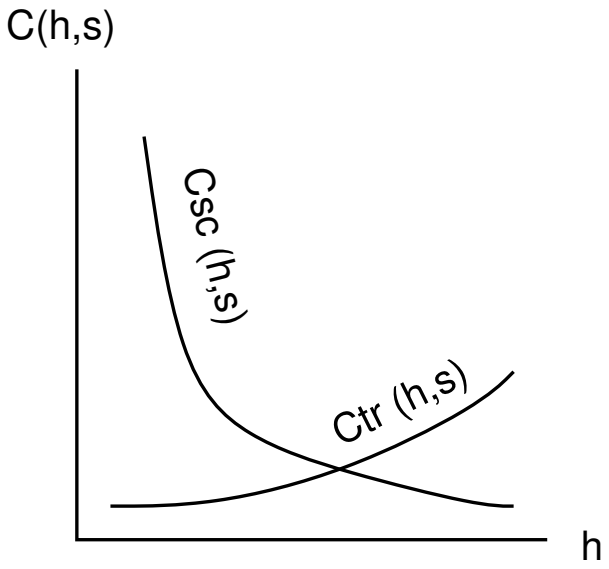


Figure 4.1: Search Structure Costs.

tual search structure, this could involve partitioning plane intersections, bounding volume tests in the internal nodes of the hierarchy, or just comparisons between ray coordinates and the partitioning planes. It also accounts for the cost involved in determining the next region along the path of the ray that is to be searched.

Other costs in ray tracing such as building the search structure and lighting calculations are not significant when compared to the total cost of rendering the scene. As we start subdividing the scene, $C_{sc}(h, s)$ decreases and $C_{tr}(h, s)$ increases. The rates of increase/decrease of these two costs will determine the performance of the search structure. In Fig. 4.1, $C_{sc}(h, s)$ and $C_{tr}(h, s)$ are two cost functions. We are interested in terminating the search structure at a height where the cost reaches a minimum.

4.1.2 Determining $C_{sc}(h, s)$

Our next step is to determine estimates for $C_{sc}(h, s)$ and $C_{tr}(h, s)$. To be of any practical use, the expressions that we obtain must be dependent on the characteristics of the scene that we are trying to render.

Let us look at $C_{sc}(h, s)$, the cost involved in examining the scene. What we want to know is, at any particular level of subdivision, what portion of the scene is examined by each ray. One way we could determine this is to try to compute the number of primitives examined by a ray on the average. So

$$C_{sc}(h, s) = \frac{C_{pr}}{n} \sum_{i=0}^{n-1} \sum_{r=0}^{R_i} n_{pr_i}(i, r, h, s)$$

where

C_{pr}	= cost of testing a primitive for intersection.
$n_{pr}(i, r, h, s)$	= number of primitives examined by ray i in region r using search structure s of height h.
n	= total number of rays spawned.
R_i	= number of regions examined by ray i.
h	= height of search structure.
s	= search structure.

Determining $n_{pr}(i, r, h, s)$ before the rendering is not easy. However, the dependency of $n_{pr}(i, r, h, s)$ on region r can be removed by approximating it by an average region primitive count.

$$\begin{aligned} C_{sc}(h, s) &= \frac{C_{pr}}{n} \sum_{i=0}^{n-1} \sum_{r=0}^{R_i} n_{pr}(i, h, s) \\ &= \frac{C_{pr}}{n} \sum_{i=0}^{n-1} R_i n_{pr}(i, h, s) \end{aligned}$$

where

$n_{pr}(i,h,s)$ = average number of primitives per region.

R_i, the number of regions examined by ray i, is also difficult to obtain prior to rendering the scene. Again, approximate R_i by R, the average number of regions traversed by a ray before it terminates.

$$\begin{aligned} C_{sc}(h,s) &= \frac{C_{pr}}{n}\sum_{i=0}^{n-1} R_i n_{pr}(i,h,s) \\ &= \frac{C_{pr}}{n}\sum_{i=0}^{n-1} R n_{pr}(h,s) \\ &= \frac{C_{pr}}{n} n R n_{pr}(h,s) \\ &= C_{pr} R n_{pr}(h,s) \end{aligned}$$

where $n_{pr}(h,s)$ = average number of primitives in a region of the search structure s.

n_{pr} can be determined by examining the regions of the search structure containing collections of object primitives. The only other unknown quantity, R, the expected number of regions examined by each ray, will be estimated as follows.

In determining R, we must bear in mind that the intersection search is ordered along the path of the ray. Once an intersection is found, processing stops for that particular ray. How quickly this might happen depends on the scene complexity, in terms of how dense or space filling the primitives in the scene are. Fig. 4.2 illustrates a ray starting at its origin O and being traced in the direction $\vec{d}$. Regions 1 through 4 are in the path or close to the ray and will be examined in this order. The question is, how many of these regions will be examined? For this, we need some knowledge of the probability of a ray-primitive intersection in each of these regions.

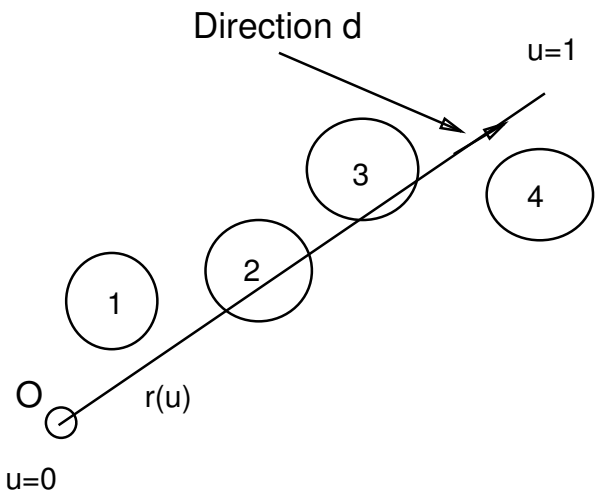


Figure 4.2: Ray Traversal.

Let p_j represent the probability that the ray has an intersection with a primitive in region j. Then $(1 - p_j)$ will be the probability that the ray will not intersect any primitive in region j. Also, let us assume that the ray is within the scene of interest, so that at least one region must be examined for intersection. So

P(1 region will be examined) = 1

P(2 regions will be examined) = $(1 - p_1)$

P(3 regions will be examined) = $(1 - p_1)(1 - p_2)$

.....

.....

P(k regions will be examined) = $(1 - p_1)(1 - p_2)....(1 - p_k)$

The expected number of regions examined is then

$$R \;=\; MAX(1, \sum_{i=1}^{k} i p_i \prod_{j=1}^{i-1}(1 - p_j))$$

Again, we can use an average region probability instead of the $p_j s$.

Let this probability be p. p is a weighted average, accounting for the different sizes of the regions. The above expression becomes

$$\begin{aligned} R &= MAX(1, \sum_{i=1}^{k} ip \prod_{j=1}^{i-1}(1-p)) \\ &= MAX(1, \sum_{i=1}^{k} ip(1-p)^{i-1}) \\ &= MAX(1, \sum_{i=0}^{k} ip(1-p)^{i-1}) \end{aligned}$$

A closed form solution to the sum of this series can be derived as follows.

$$\begin{aligned} & \sum_{i=0}^{k} ip(1-p)^{(i-1)} \\ = \; & p\sum_{i=0}^{k} i(1-p)^{(i-1)} \\ = \; & -p\sum_{i=0}^{k} \frac{d}{dp}(1-p)^{i} \\ = \; & -p\frac{d}{dp}\sum_{i=0}^{k} (1-p)^{i} \\ = \; & -p\frac{d}{dp}\left[\frac{1-(1-p)^{k+1}}{1-(1-p)}\right] \\ = \; & \frac{1}{p}\left[\{1-(1-p)^{k+1}\} - p(k+1)(1-p)^{k}\right] \end{aligned}$$

For large k,

$$\begin{aligned} \lim_{k->\infty} (1-p)^{k+1} &= 0, \qquad \text{and} \\ \lim_{k->\infty} (k+1)(1-p)^{k} &\approx \lim_{k->\infty} k(1-p)^{k} = 0 \end{aligned}$$

Therefore,

$$\begin{aligned} R &= MAX\left(1, \frac{1}{p}\left[\{1-(1-p)^{k+1}\} - p(k+1)(1-p)^k\right]\right) \\ &\approx MAX(1, 1/p) \\ &= 1/p \end{aligned}$$

4.1.3 Determining Region Probability p

The average region probability p is the probability that a primitive in a region will be intersected by an incoming ray. Determining this accurately is very expensive and might be impossible since it depends on the geometry of the region, the primitives in it and the ray distribution. We need to find a reasonable approximation.

The region probability can be estimated by enclosing the collection of primitives by a convex bounding volume. Since most space subdivision methods produce convex partitions, the regions are already convex. The ratio of primitives' bounding volume surface area to that of the region gives an estimate of the conditional probability, using Theorem 3.1. A more accurate value of p can be obtained by enclosing individual primitives with bounding volumes, thus accounting for the void space between primitives. However, if two bounding volumes overlap, then the overlap area has to be subtracted out since it cannot be counted twice. In our implementation, we use the ratio of bounding volume surface areas to estimate the conditional probability. Each of the region probabilities must be weighted by the region size (again, the region surface area can be used), when we compute the average probability. If A_i represents the bounding volume surface area and E_i the surface area of the extent of the same region ($A_i \leq E_i$, since the tightest bounding volume of the primitives clipped to to extent is usually less than the surface area of the

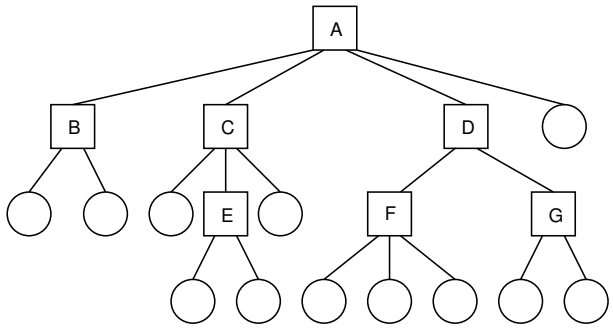


Figure 4.3: A Bounding Volume Hierarchy

extent), then the region probability is given by

$$\begin{aligned} p &= \frac{\sum_{i=1}^{n} E_i * \frac{A_i}{E_i}}{\sum_{i=1}^{n} E_i} \\ &= \sum_{i=1}^{n} A_i/E_i \end{aligned}$$

where n is total number of leaf node regions in the search structure.

4.1.4 Modification for Bounding Volume Hierarchies

In bounding volume hierarchies, the traversal of the hierarchy is usually not along the path of the ray (refer to Section 3.3.1 and Table 3.3.1). The expected number of regions examined by each ray, R, is calculated in a different way. Consider again the bounding volume hierarchy in Fig. 3.10. We computed the expected number of intersections (or regions examined) as

$$\begin{aligned} R &= 1 + 4 + 2P(B|A) + 3P(C|A) + \\ &\quad 2P(D|A) + 2P(E|A) + 3P(F|A) + 2P(G|A). \end{aligned}$$

where $P(I|J)$ represents the conditional probability that node I is intersected given that J has already been intersected. The conditional probabilities are

determined by using the surface area formula, as described in Equations 3.2. So the total cost is

$$C = C_{pr} R n_{pr}(h, s)$$

where C_{pr} is the cost of a bounding volume test and $n_{pr}(h, s)$, the average number of primitives at each leaf node. In Goldsmith's method, $n_{pr}(h, s) = 1$. The above equation represents the total cost. Separating this into the scene cost and traversal cost, we get

$$\begin{aligned} C_{sc} &= 1 + 3 + 1P(C|A) + 2P(D|A) \\ C_{tr} &= 1 + 2[P(B|A) + 2P(C|A) + 2P(E|A) + 2P(G|A)] + 3P(F|A) \end{aligned}$$

4.1.5 Determining $C_{tr}(h, s)$

The traversal cost $C_{tr}(h, s)$, in general, is given by the following form:

$$C_{tr}(h, s) = R * C_r(h, s)$$

where

R = expected number of regions examined by the ray

$C_r(h, s)$ = average traversal cost expended per region.

Thus, the higher R is, the more $C_{tr}(h, s)$ will be.

4.2 Validating the Model

We would next like to investigate how the costs computed using this model compare to those obtained from ray tracing of real scenes. We have implemented the uniform subdivision method, the Kaplan-BSP tree, octree, and the ABV hierarchy. For each of these structures, the model is evaluated for

Parameter	Parameter Values		
	Model	Experimental	
		With BG Rays	Excluding BG Rays
Cost(floats/ray)	463.8	197.8	283.2
Hitting Prob.	0.189	0.285	0.199
Bound Vol. Test Cost	15.5	15.58	
Plane Test Cost	3.5	3.6	
Avg. Height	9.02	9.17	
Avg. Leaf Count	3.63	2.78	

Table 4.1: Tetra (BSP)

each value of the maximum height of the structure (for the uniform subdivision method, the resolution of the grid is doubled in all three dimensions). The scene is ray traced for each value of the maximum height. A record of the total number of bounding volume tests and partitioning plane tests, which dominate the computation of ray tracing is recorded. All costs are normalized to floating point operations. Both the predicted cost (from the cost model) and the actual cost (from the ray tracing) are plotted against the maximum height.

4.2.1 Comparing Model Parameters with Experimental Parameters

Figures 4.4, 4.5 and 4.6 illustrate the predicted and actual cost characteristics for different search structures on several scenes. There are two important features to notice in these characteristics. First, the general behavior (or shape) of the predicted characteristics matches quite well with the actual (experimental) characteristics, showing that the cost model is tracking the actual characteristics as a function of the maximum height. This will be the key factor in making this model a very useful tool for providing automatic termination criteria for ray tracing search structures, as we will demonstrate in the following chapter.

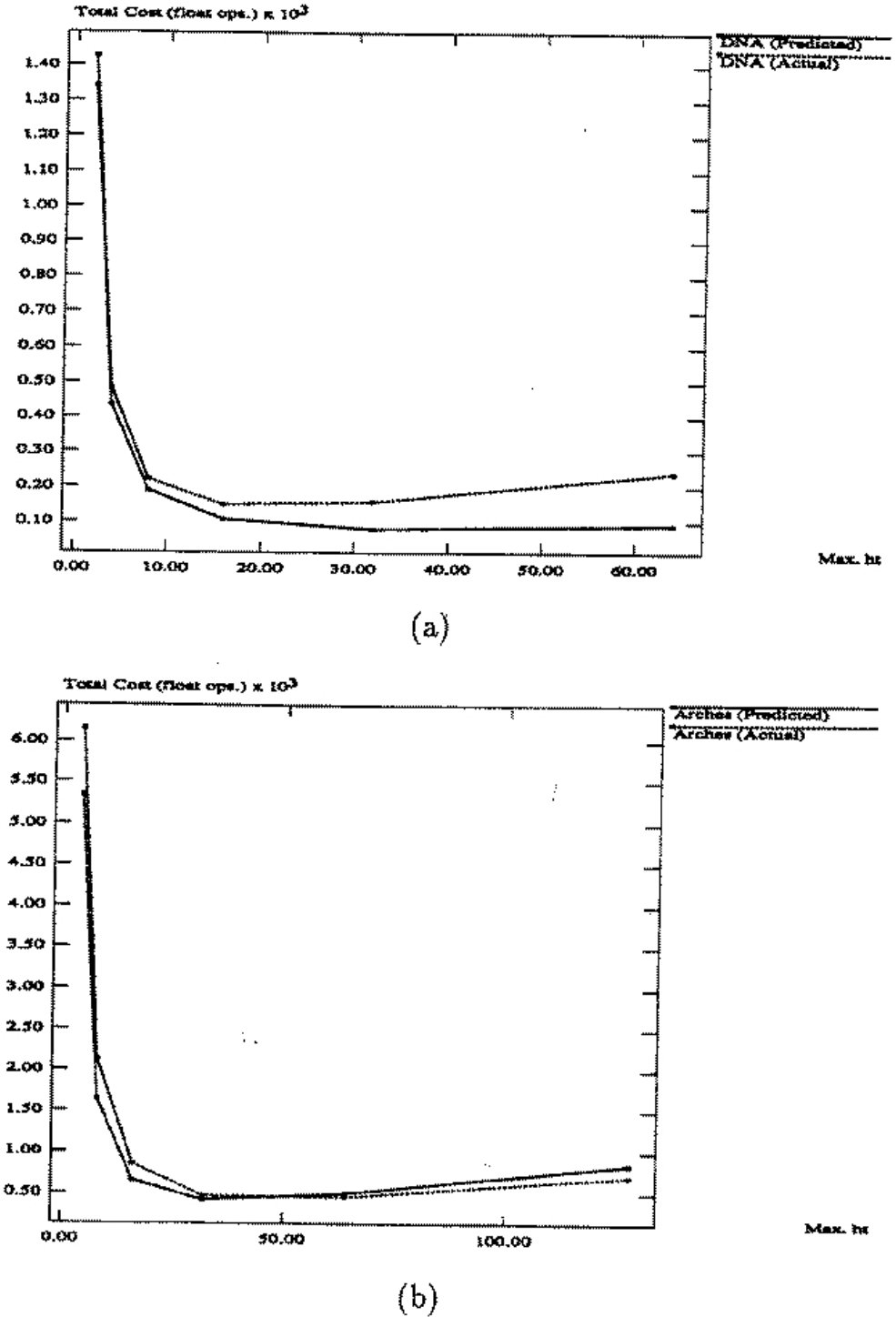
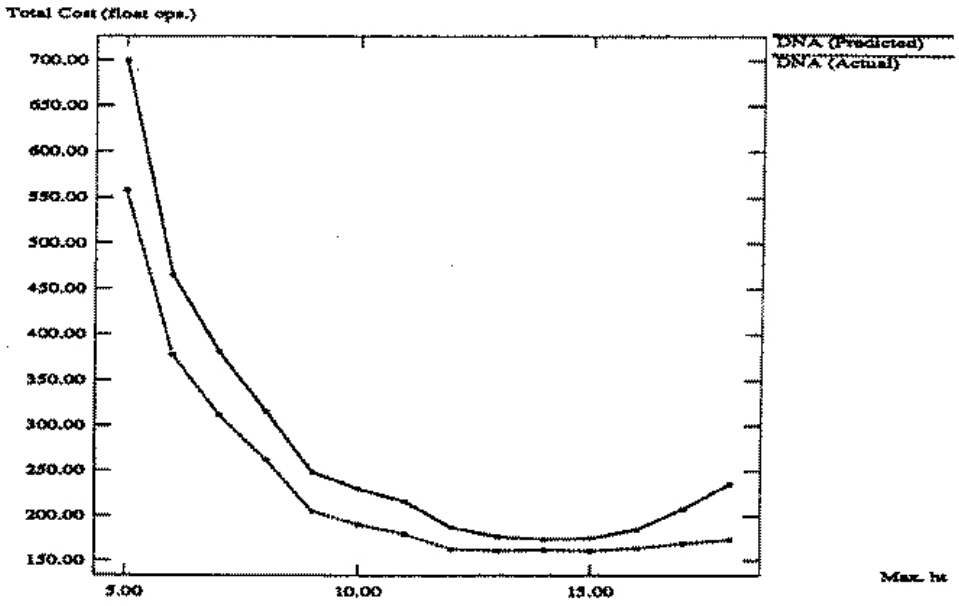
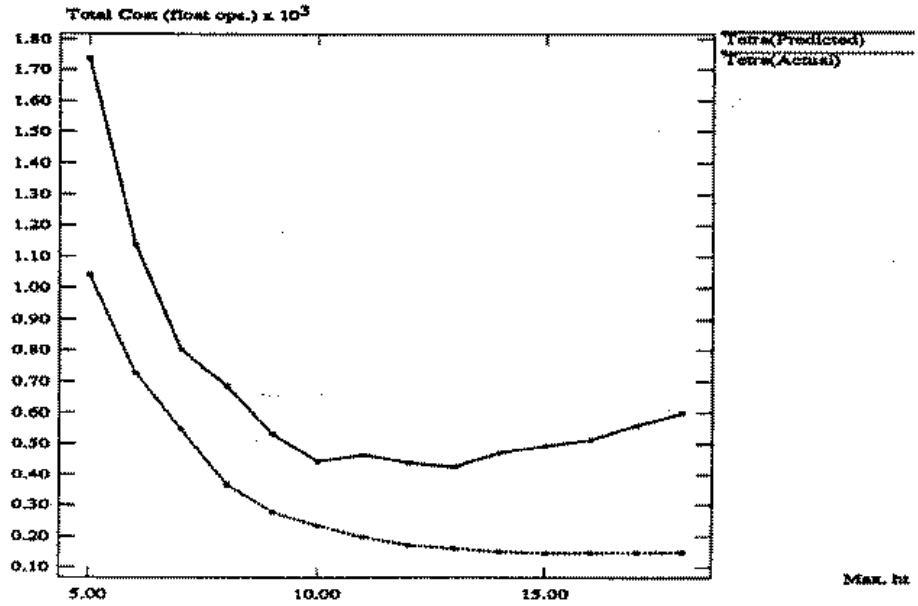


Figure 4.4: Unif. Subd. Method Characteristics (a) DNA (b) Arches.

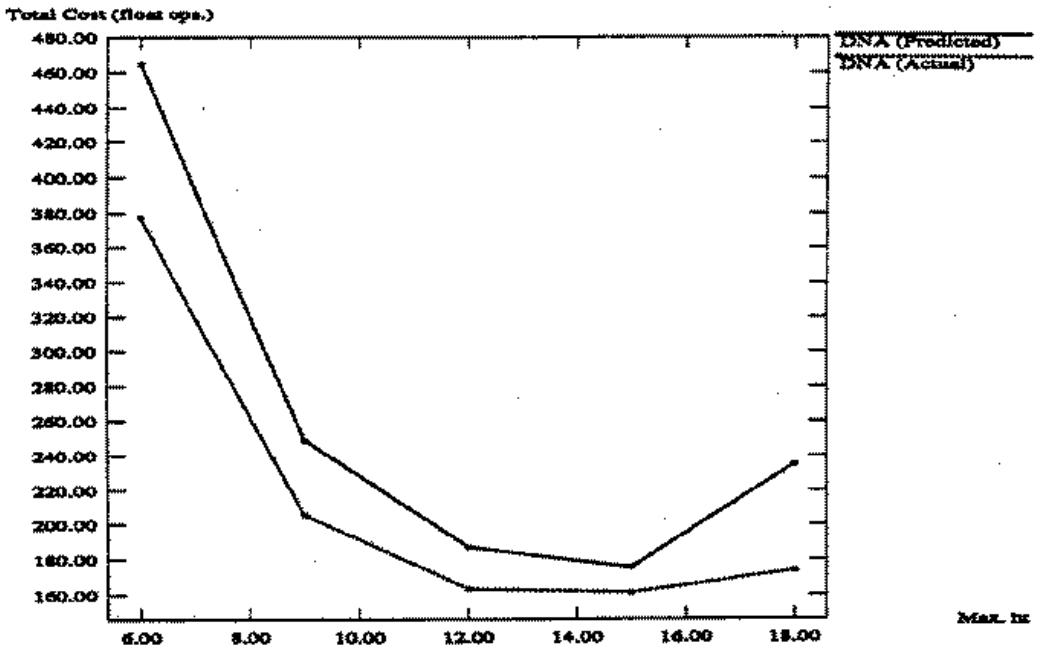


(a)

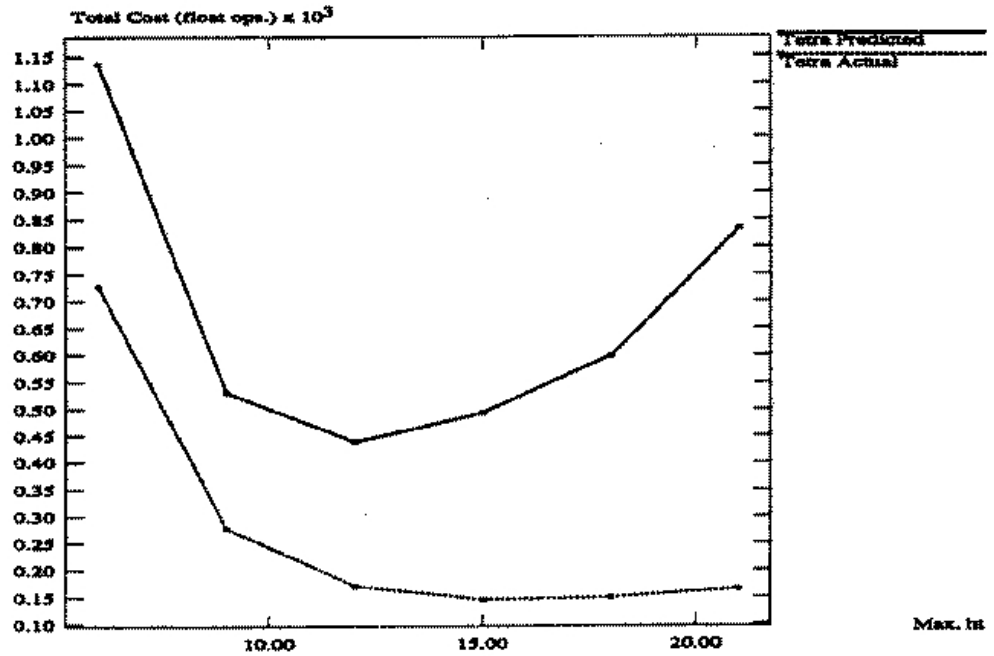


(b)

Figure 4.5: BSP Tree Characteristics (a) DNA (b) Arches.



(a)



(b)

Figure 4.6: Octree Characteristics (a) DNA (b) Tetra.

Parameter	Parameter Values		
	Model	Experimental	
		With BG Rays	Excluding BG Rays
Cost(floats/ray)	248.7	214.3	276.4
Hitting Prob.	0.45	0.354	0.274
Bound Vol. Test Cost	15.5	17.13	
Plane Test Cost	3.5	3.55	
Avg. Height	8.6	8.61	
Avg. Leaf Count	5.28	3.616	

Table 4.2: DNA (BSP)

Parameter	Parameter Values		
	Model	Experimental	
		With BG Rays	Excluding BG Rays
Cost(floats/ray)	753.1	356.3	522.4
Hitting Prob.	0.102	0.16	0.109
Bound Vol. Test Cost	15.5	16.35	
Plane Test Cost	3.5	3.58	
Avg. Height	11.44	12.18	
Avg. Leaf Count	2.37	3.18	

Table 4.3: Arches (BSP)

Parameter	Parameter Values		
	Model	Experimental	
		With BG Rays	Excluding BG Rays
Cost(floats/ray)	127.33	519.3	519.3
Hitting Prob.	0.297	0.074	0.074
Bound Vol. Test Cost	15.5	13.58	
Plane Test Cost	3.5	3.6	
Avg. Height	7.67	15.84	
Avg. Leaf Count	0.71	2.10	

Table 4.4: Spd.Balls (BSP)

Parameter	Parameter Values		
	Model	Experimental	
		With BG Rays	Excluding BG Rays
Cost(floats/ray)	261.9	140.4	201.1
Hitting Prob.	0.08	0.122	0.085
Bound Vol. Test Cost	15.5	15.83	
Cell Test Cost	6.5	6.8	
Avg. Leaf Count	0.94	0.667	

Table 4.5: Tetra (Uniform Subdivision)

Second, the predicted and actual cost values do not match very well in several of these cases. To understand this better, let us look at some examples to compare the cost parameters between the model and those obtained from the experiments. This will help us understand better the reason why the absolute values of the predicted costs do not match well with the experimental values.

Tables 4.1 through 4.7 illustrate the parameter values from the cost model and those obtained from the experiment. For experimental values, some of the parameters are illustrated with and without 'BG Rays', which refer to rays that completely miss the scene extent. Tables 4.1 through 4.4 illustrate four cases using the Kaplan-BSP method and Tables 4.5 through 4.8 illustrate the same cases using the uniform subdivision method. All costs are measured in floating point operations. Here 'Cost (floats/ray) refers to the expected cost of tracing each ray. 'Hitting Prob' is the probability that a ray penetrates a leaf node region (containing object primitives) given that a ray enters its parent region. 'Avg. Leaf Count' is the average number of object primitives encountered at any leaf node of the hierarchy. For the uniform subdivision method, leaf nodes are simply the grid voxels. 'Cell Test Cost' refers to the work done in moving from one voxel to the next.

Parameter	Parameter Values		
	Model	Experimental	
		With BG Rays	Excluding BG Rays
Cost(floats/ray)	106.72	114.4	147.6
Hitting Prob.	0.301	0.172	0.133
Bound Vol. Test Cost	15.5	17.2	
Cell Test Cost	6.5	6.53	
Avg. Leaf Count	1.65	0.758	

Table 4.6: DNA (Uniform Subdivision)

Parameter	Parameter Values		
	Model	Experimental	
		With BG Rays	Excluding BG Rays
Cost(floats/ray)	421.8	320.9	470.5
Hitting Prob.	0.033	0.063	0.043
Bound Vol. Test Cost	15.5	15.72	
Cell Test Cost	6.5	6.89	
Avg. Leaf Count	0.49	0.86	

Table 4.7: Arches (Uniform Subdivision)

Parameter	Parameter Values		
	Model	Experimental	
		With BG Rays	Excluding BG Rays
Cost(floats/ray)	484.6	1128.3	1128.3
Hitting Prob.	0.016	0.023	0.023
Bound Vol. Test Cost	15.5	12.53	
Cell Test Cost	6.5	6.91	
Avg. Leaf Count	0.07	1.48	

Table 4.8: Spd.Balls (Uniform Subdivision)

The total cost is measured by counting up the operations performed for bounding volume tests, plane tests and those required to move from voxel to voxel. The expected number of regions visited by each ray is measured by counting up the total number of leaf node regions (or voxels for uniform subdivision) visited by all rays and dividing this by the total number of rays. The reciprocal of this figure gives the hitting probability. In a similar manner, the average leaf node object count is determined.

The first important point to notice in all of these tables is the improvement in predictions in the absence of background rays (except the Spd.Balls case which has no background rays). This is not surprising since the cost model was developed assuming that each ray penetrated the input scene extent. The total cost and the hitting probability values predicted by the cost model are closer to the experimental values when background rays are excluded.

The bounding volume test cost, cell test cost and the plane test costs match quite well in all the cases.

The predictions are significantly off in the case of the Spd.Balls case (Table 4.4). Both the hitting probability and the expected height are far from the model predictions, resulting in a significant mismatch in the total cost. For the uniform subdivision method (Table 4.8), it is the voxel object count that causes the mismatch. All the rays in this model enter the bounding volume of the scene. Thus, there are no 'BG rays' as in the other test cases.

Two different factors could account for the big mismatch in the Spd.Balls case. The experimental value of the hitting probability shown in Tables 4.1 through 4.8 takes into account the void spaces within the object clusters at the leaf nodes of the hierarchy (for the uniform subdivision method, the object clusters at each voxel of the grid). However, the cost model does

Scene	Total Cost		Hitting Prob.
	Model	Experimental	
Tetra	463.8	343.8	0.17
Dna	248.7	280.4	0.27
Arches	753.1	312.1	0.17
Spd.Balls	127.33	99.6	0.25

Table 4.9: Experiments Using Random Ray Distribution

not account for this (refer to Section 4.1.3).

The experimentally determined hitting probability takes into account the void spaces within object clusters at the leaf nodes (or voxels). To make sure that void spaces within object clusters are not taken into account (and thus trying to see if it behaves like our model) all rays that enter the bounding volumes of the object clusters are ignored. This will cause the hitting probability to increase, since a ray is now assumed to intersect an object primitive if it penetrates the bounding volume containing it.

With this modification, for the Spd.Balls case, the hitting probability increased from 0.074 to 0.086. This is still significantly off from the model value of 0.297. If we had a way to take care of void spaces within object clusters, this would result in bringing the predicted cost a little closer to the experimental value.

Secondly, the hitting probabilities are computed under the assumption that rays are uniformly distributed. We would also like to see experimentally if the distribution of the rays used in the ray tracing is sufficiently close to being uniform.

To verify this, a collection of rays with random origins (but located within the scene extent) and directions were traced for each scene. In our

experiment, we spawned 20,000 rays in this manner and collected the same statistics as before. The results are shown in Table 4.9. The biggest difference is in the results for the Spd.Balls case. The expected cost is much closer to the model value, when compared to the figures in Table 4.4. This tells us that the distribution of rays used in the ray tracing for this particular scene was far from uniform, unlike the DNA case whose cost is almost the same as before. The hitting probability for Spd.Balls case was 0.25, a big improvement from 0.074. The figures for the Tetra are also better, but this is partly because the hitting probability value dropped a little. The mismatch for the Tetra and DNA cases are principally because of the average leaf object counts. All other parameter values are fairly close. For the arches case, it is the hitting probability that is causing the mismatch.

4.3 Conclusions

In this chapter, we have presented a cost model for search structures commonly used in ray tracing. It is computed from the statistical characteristics of the search structures that are being used to accelerate ray tracing. The model is inexpensive to compute and can be done during a preprocessing step, when the search structure is being built, as we will show in the next chapter. The model has been evaluated on some of the common search structures being used in ray tracing. The model characteristics have been shown to track the experimental characteristics as the height of the search structure is varied. The assumptions made in developing the cost model causes a mismatch in the absolute values of the predicted costs and the actual costs. The presence of 'BackGround' rays and the uniform ray distribution assumption are two different factors that have been shown to cause this mismatch.

The model as presented here is not very useful for predicting the computational requirements for ray tracing a given scene. However, in the next chapter, we will show that the model is useful in comparing different methods for ray tracing a given scene. In addition, it will be shown to be a very effective means of predicting termination criteria for these search structures.

Chapter 5

APPLICATIONS OF THE COST MODEL

Search structures such as the BSP tree, octree, uniform subdivision and nested bounding volumes are all being used with great success. However, previous work has not revealed how deep the hierarchical data structures (or how much subdivision in the uniform subdivision technique) should be constructed to achieve optimum results. Heretofore termination criteria for hierarchy construction algorithms have been totally ad hoc, leaving open the question of whether methods employing hierarchical search structures have really been exploited to their full potential.

Performance evaluations of these structures through the use of timing benchmarks show that no search structure seems to be best on all scenes. Thus, if an idea of the performance of a search structure on a given scene is known before rendering, then a decision could be made as to use it or substitute it with another search structure.

In this chapter, we demonstrate two important applications of the cost model we developed in the previous chapter; determining automatic termination criteria for ray tracing hierarchies, and in choosing a search structure for a scene for providing the best performance. In both of these applications, the model evaluation can be built into the preprocessing step. For the hierarchy termination problem, the model is evaluated as the hierarchy is being constructed so that it may be terminated when the cost reaches a minimum.

For choosing a search structure, it is necessary to build several of these search structures and determine their costs using the cost model. The structure with the smallest cost will then be used for the ray tracing.

5.1 Automatic Termination Criteria for Ray Tracing Search Structures

5.1.1 The Problem

We will be concerned with several of the common search structures being used in ray tracing, including BSP trees [25], octrees [17], bounding volume hierarchies [39][18] and the uniform subdivision method. Among BSP trees, we will specifically be concerned with the structure used by Kaplan for ray tracing. We will begin with a description of the hierarchy termination problem associated with each of these methods.

As we described in chapter 3, Kaplan's [25] implementation of the BSP tree uses axis-aligned planes to recursively partition space. The subdivision continues until the subdivided nodes contain either a small number of primitives or their size becomes smaller than a set threshold. In order to optimize performance, this threshold must be set correctly. If the threshold is too high, then large numbers of primitives end up in each voxel; if it is too low, getting to the leaf nodes from the root node of the tree is more expensive.

The octree hierarchy used by Glassner [17] performs a subdivision identical to Kaplan's BSP tree. Each level of the octree corresponds to three levels of the BSP tree. The termination problem in the octree is thus the same as in the BSP tree.

The uniform subdivision method subdivides space like the octree, but the subdivision is uniform, resulting in a 3-d grid of equal sized voxels. The

non-adaptive property of this structure can potentially result in large regions of empty voxels, which will be expensive to traverse. However, restricting the subdivision can also put large numbers of primitives in some of the voxels. Thus, it is difficult to know the correct amount of subdivision without any knowledge of the scene characteristics.

In the previous chapter, we showed that the model used by Goldsmith's ABV hierarchy is a special case of the cost model that we developed. We will apply the cost model to the ABV hierarchy, which is the best known bounding volume hierarchy to date, principally because it is built automatically, which is very convenient for scenes containing large numbers of primitives.

5.1.2 Evaluating the Cost Model

We will next describe how to use the cost model in helping us terminate a search structure. The two costs that need to be computed during the preprocess are

$$\begin{aligned} C_{sc}(h,s) &= C_{pr} R n_{pr}(h,s) \\ C_{tr}(h,s) &= R C_r(h,s) \end{aligned}$$

All the terms in the above expression can be computed. C_{pr} is taken as the cost of intersecting the bounding volume around an object primitive. This is a reasonable approximation since quite a few of the rays will miss the bounding volume, in which case the primitive inside need not be examined. $C_{sc}(h,s)$ is typically an exponential function of the height of the hierarchy. R is computed as described in the previous chapter. $n_{pr}(h,s)$ can be determined by summing up the primitive counts at the leaf nodes of the hierarchy. The leaf node counts are weighted by the size of their regions. The bounding volume surface area

of the region is used to weight the primitive counts. Lastly, $C_r(h, s)$, has to be calculated. For this, the cost of reaching the leaf nodes from the root has to be determined. This is found by multiplying the work done per node by the average height of the hierarchy. The average height of the hierarchy is also a weighted quantity, since paths leading to nodes which are larger in size will be visited more often by rays when compared to paths leading to smaller sized regions. For the uniform subdivision method, $C_r(h, s)$ is determined by the work done in moving from the current voxel to the next voxel visited by the ray.

As we start building the hierarchy, the two costs are computed for each value of the maximum height. Initially, the decrease in $C_{sc}(h, s)$ will overwhelm the increase in $C_{tr}(h, s)$. What we are looking for is the point where the cost reaches a minimum. To determine this, we may have to build the hierarchy a few extra levels to make sure we have reached the minimum. This increases the cost of the preprocess. In general, this is not very significant when compared to the total rendering time, especially when the environments are very complex. It is also possible that there might be several local minima (for example, the BSP and octree structures exhibit this characteristic). If these are all close to each other, we could pick any one of the minima with little difference on performance.

5.1.3 Implementation and Experimental Results

We have implemented the BSP tree, octree, uniform subdivision and the ABV Hierarchy methods to test the cost model. All experiments were

Ray counts are in thousands

Scene	*Tetra*	*DNA*	*Arches*	*Balls*	*Geo58*
Objects	*1024(P)*	*410(S)*	*4818(P)*	*7382(PS)*	*306(P)*
Lights	*1*	*1*	*2*	*3*	*3*
Total rays	*299*	*445*	*437*	*1819*	*847*
Visual rays	*262*	*323*	*317*	*722*	*392*
Shadow rays	*37*	*122*	*120*	*1097*	*455*
Hit rays	*44*	*75*	*73*	*873*	*278*

Table 5.1: Statistics of Test Scenes.

conducted on a Sun 4/280 workstation running SunOS UNIX[1] 4.0.3. Five different data sets were used as test cases. Details of these data sets are given in Table 5.1. P stands for polygons and S for spheres. 'Objects' refer to the total primitive objects in the scene, 'Lights' refer to the total light sources in the scene, all of which are point sources. 'Visual Rays' are rays spawned from eye position (primary rays), 'Total rays' include primary and all higher generation rays and rays to light sources for computing shadows, 'Shadow' rays are rays spawned toward light sources for computing shadows. 'Hit rays' are the total number of rays that intersected some object primitive in the scene.

Images of these models are shown in the color plates of chapter 9. The termination parameter used in all these methods except uniform subdivision is the maximum height. For the uniform subdivision method, the termination parameter is the grid resolution n_grid.

Before we get into the results for each method, some general notes need to be made. In our implementation, each primitive is surrounded by a bounding volume which is an axis-aligned parallelepiped. Bounding volumes around collections of primitives are also axis-aligned parallelepipeds. In our

[1]UNIX is a trademark of AT&T Bell Laboratories.

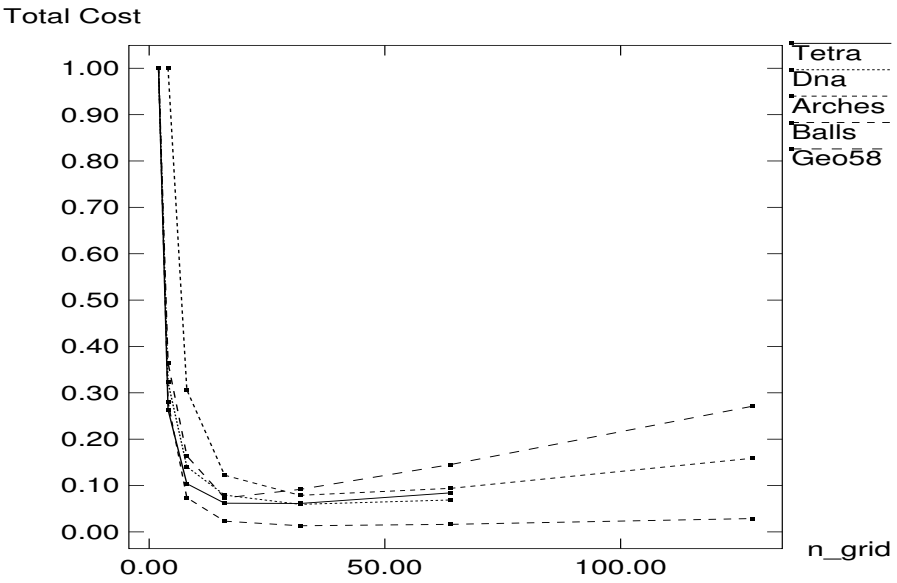
implementation $C_{pr} = 15.5$ floating point operations. In the BSP tree, octree, and uniform subdivision methods, the search can be stopped prematurely once an intersection is found because objects are examined along the path of the ray, starting from its origin. This is not true in bounding volume hierarchies, although a partial ordering can be obtained by sorting the bounding volume intersection points obtained during ray tracing. Duplicate object intersection tests are avoided by maintaining ray signatures in all object records. An object is tested for intersection only if it has not been examined by the current ray.

For each method, we plot both predicted and actual performance characteristics as a function of the termination parameter. For the predicted characteristic, the total cost = C_{sc}+C_{tr}. This is plotted against the termination parameter. The actual (or experimental) characteristic is a plot of the running time (which is a measure of the total cost) versus the termination parameter. The total cost has been normalized from 0.0 to 1.0 so as to fit all the test cases in the same graph plot.

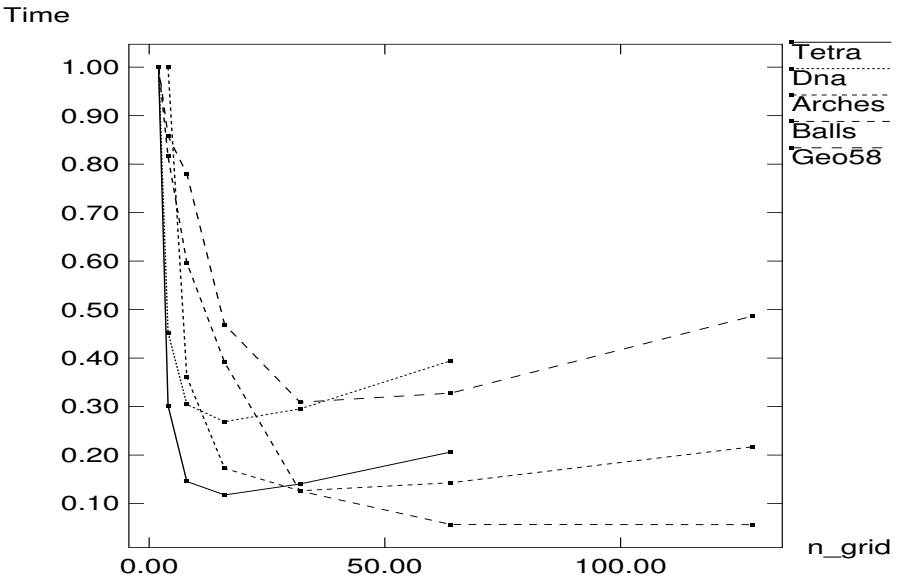
5.1.3.1 Uniform Subdivision

Our implementation of uniform subdivision method follows the algorithms outlined in [6] and described in chapter 3. In our implementation, it takes about 6.5 floating point operations to move from one voxel to the next, on the average. Thus $C_r = 6.5$.

Fig. 5.1 and Table 5.2 show the performance characteristics and results for the uniform subdivision method. Here n_grid is the resolution of the grid in each of the three dimensions. For testing the cost model, the resolution is doubled in all three dimensions each time we subdivide. In our implementation, we used polyhedral bounding boxes [26] to enclose the primitives when



(a)



(b)

Figure 5.1: Unif. Subd. Method Characteristics (a) Predicted (b) Actual.

Scene	Optimal Height		Time(min.)
	Predicted	Actual	
Tetra	16,32	16,32	3.04
DNA	32	16	5.16
Arches	32	32	8.37
Balls	32	64	92.0
Geo58	16	32,64	22.47

Table 5.2: Unif. Subd. Predictions.

clipping it to the voxels. The clipped points were used in computing an axis-aligned bounding box within the voxel. The surface area of this box was used in computing the region probability. More accurate methods of determining the surface area of the primitive within the voxel will improve the predictions.

5.1.3.2 BSP Tree

Our implementation of the BSP tree hierarchy is very similar to that of Kaplan's [25]. One difference is that each subdivision step does not necessarily produce eight octants. If for example, a plane has subdivided the original space into two equal sized voxels and one of them does not contain any primitives, then it is not subdivided by the remaining two planes. This results in a smaller number of empty regions, thus making the structure more adaptive to the scene.

The traversal method used in our implementation is the k-d tree traversal, which we will describe in the next chapter (Section 6.2.5). For now, it is to be noted that it identifies the regions along the path of the ray just as the BSP/octree traversal, but is computationally more efficient. $C_r = 4.5$ in our implementation, on the average. This has to be multiplied by the av-

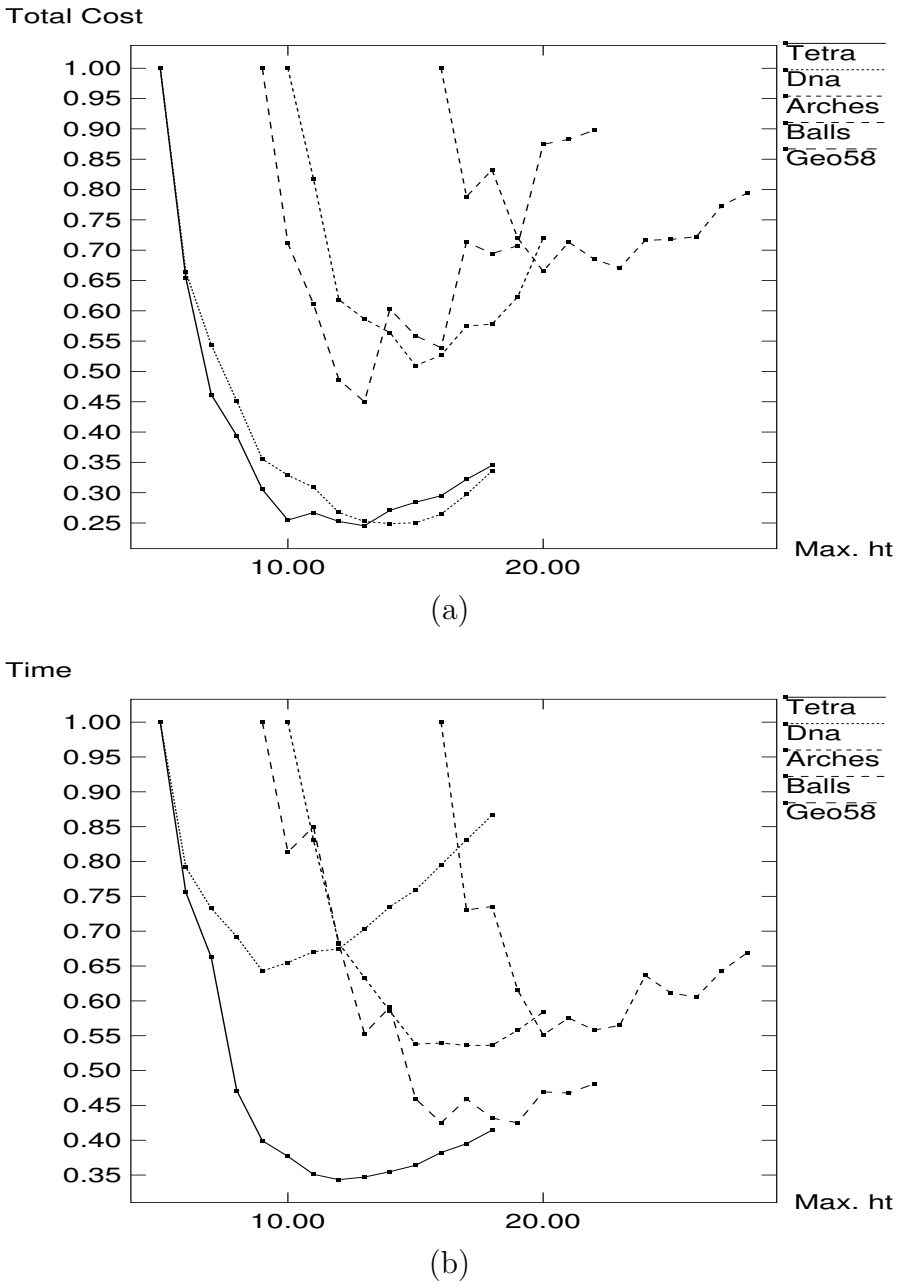


Figure 5.2: BSP Tree Characteristics (a) Predicted (b) Actual.

Scene	Optimal Height		Time(min.)
	Predicted	Actual	
Tetra	13	11-13	3.50
DNA	13-15	9,10	6.41
Arches	15	15-19	9.64
Balls	20,23	20,22,23	67.61
Geo58	13	16,18,19	25.95

Table 5.3: BSP Tree Predictions.

erage height of the hierarchy, to account for the work done to reach the leaf nodes. Fig. 5.2 shows the performance characteristics for the BSP tree structure for several scenes. Table 5.3 illustrates the results of using the cost model to predict the height at which the subdivision has to be stopped for optimal performance. Overall, the predicted heights are very close to the optimal heights obtained by the experiments. In general, for the BSP tree structure, there is a small range of heights at which the performance stays relatively constant. It is interesting to note that in several of these cases, optimal performance occurs well beyond the logarithm of the number of objects, which would have been the logical choice for termination of the hierarchy, if the scene primitives were uniformly distributed. This further justifies the need for a cost model to help terminate the BSP hierarchy.

5.1.3.3 Octree

To implement the octree, we modified our BSP tree implementation so that at each step of the subdivision, eight equal sized voxels were created. The data were then collected as in the BSP tree case. The traversal method used is the same as in the BSP tree method. Fig. 5.3 and Table 5.4 show the performance characteristics and results of using the cost model. As expected, the performance is slightly worse (Balls and Geo58 scenes) than the BSP tree

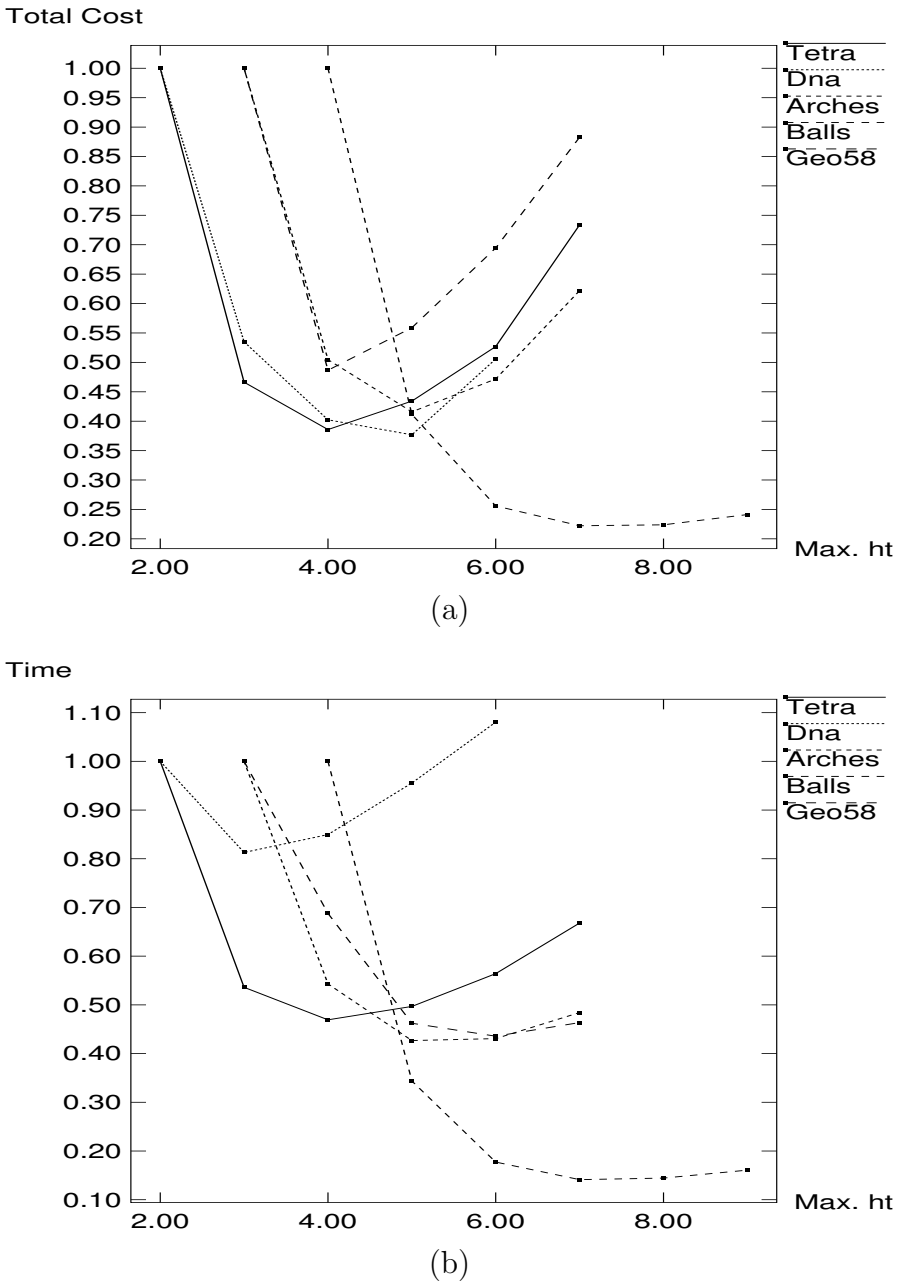


Figure 5.3: Octree Characteristics (a) Predicted (b) Actual.

Scene	Optimal Height		Time(min.)
	Predicted	Actual	
Tetra	4	4	3.5
DNA	5	3,4	6.41
Arches	5	5	9.54
Balls	7,8	7,8	74.00
Geo58	4	5,6	26.62

Table 5.4: Octree Predictions.

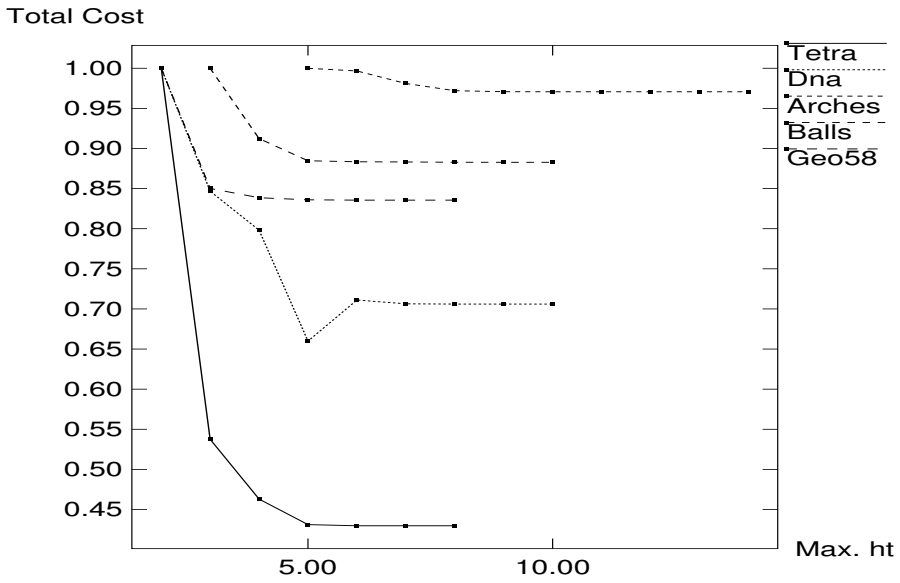
case because of the additional empty voxels that need to be processed in the octree. Note that each level of the octree corresponds to three levels of the BSP tree.

5.1.3.4 Automatic Bounding Volume Hierarchy method

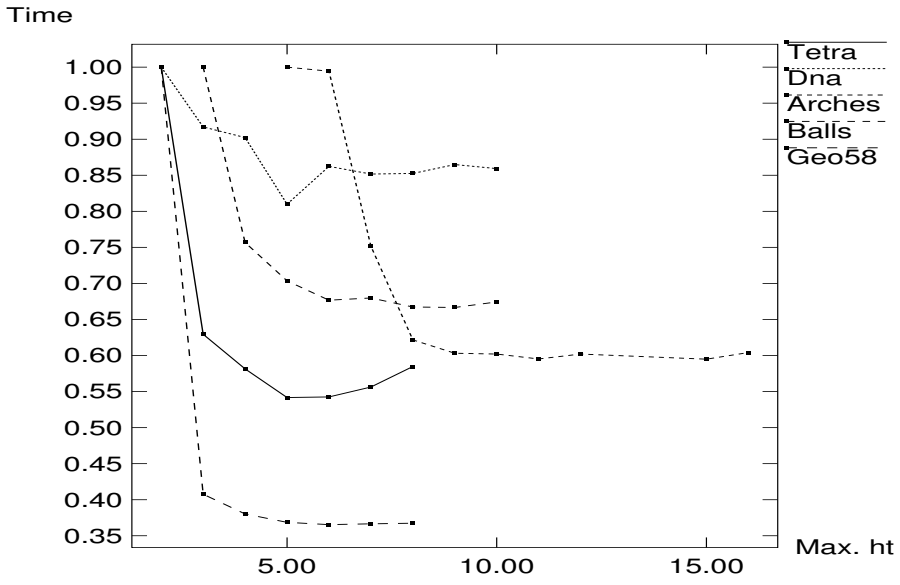
Goldsmith's automatic bounding volume hierarchy also stops building the hierarchy beyond a certain height since a heuristic search determines an insertion point in the hierarchy for each object primitive. Fig. 5.4 and Table 5.5 illustrate the characteristics and results of using the cost model. In the DNA model, there is a height at which the cost reaches a minimum, while in all the other cases, the cost remains flat after a certain height. Since all the scenes except Tetra have large numbers of secondary rays, the cost of going down the hierarchy has to be added to the predicted cost.

5.2 Choosing a Search Structure for a Given Scene

Once a search structure has been constructed, the cost model gives us an estimate of the expected number of operations performed by each ray. Since the construction of the search structure usually takes a small fraction of



(a)



(b)

Figure 5.4: ABV Hierarchy Characteristics (a) Predicted (b) Actual.

Scene	Optimal Height		Time(min.)
	Predicted	Actual	
Tetra	6	5,6	4.14
DNA	5	5	11.90
Arches	9	9	14.16
Balls	7,8	8	55.33
Geo58	6	6	16.06

Table 5.5: ABV Hierarchy Predictions.

Search Structure	Predicted Cost (float ops./ray)	Time(min.)
Unif. Subd.	261.9	3.1
BSP	426.3	3.5
Octree	438.7	3.5
ABV	669.8	4.14

Table 5.6: Tetra

the total rendering time, it is possible to build several of these search structures and choose the one with the smallest estimated cost. We will proceed to do this and show how the cost model is effective in choosing a search structure for a given scene.

Search Structure	Predicted Cost (float ops./ray)	Time(min.)
Unif. Subd.	79.8	8.4
BSP	174.2	6.4
Octree	175.2	6.4
ABV	884.8	11.9

Table 5.7: DNA

Search Structure	Predicted Cost (float ops./ray)	Time(min.)
Unif. Subd.	421.7	8.4
BSP	778.4	9.6
Octree	753.1	9.5
ABV	941.5	14.2

Table 5.8: Arches

Search Structure	Predicted Cost (float ops./ray)	Time(min.)
Unif. Subd.	393.1	92.0
BSP	128.4	67.6
Octree	147.1	74.0
ABV	562.0	55.3

Table 5.9: Spd.Balls

5.2.1 Experimental Results

Tables 5.6, 5.7, 5.8, 5.9 and 5.10 show the predicted costs (from the cost model) and run times (from the ray tracing) for each scene using the five different search structures. There are several important points to be noted from these statistics.

- A look at the results shows that there is no one structure that performs consistently better than the others. In the Tetra and Arches models, the

Search Structure	Predicted Cost (float ops./ray)	Time(min.)
Unif. Subd.	125.9	22.5
BSP	174.7	25.6
Octree	157.7	26.6
ABV	128.6	16.1

Table 5.10: Geo58

uniform subdivision method performs better than the BSP, Octree and ABV hierarchies. In the DNA model, the BSP and Octree structures perform better than the uniform subdivision and ABV structures. In the Spd.balls and Geo58 models, the ABV structure wins over the BSP, Octree and Uniform Subdivision methods.

- The cost model predicts the relative performance of each of the search structures for the Tetra and Arches scene models. In the remaining cases, the model makes a wrong choice. If the BSP/Octree methods had been chosen, the performance for the DNA scene would have been about 23% better. For the Spd.Balls model, if the ABV hierarchy had been chosen, the performance would have improved by 18%. In the Geo58 model, if the ABV hierarchy had been the choice, the performance would have improved by 28%.

  However, using the model is better than not using it all. In the DNA model, the ABV hierarchy might very well have been the choice, which takes almost twice as long to render the scene when compared to the BSP/Octree methods. Similarly, for the Spd.Balls model, if uniform subdivision had been used, the performance would have suffered by nearly 36%.

- While the choice of search structure to be used is being made with reasonable accuracy, it must be noted that the absolute values of the costs are not quite proportional to the run times. Such a result would be useful in estimating the rendering time and help one allocate computational resources for a particular application. More sophisticated methods of estimating the model parameters would bring us closer to this goal.

5.3 Conclusions

In this chapter, we have presented two important applications of the cost model that we developed in the previous chapter. The model has been shown to provide an effective means of building automatic termination criteria into subdivision techniques such as the BSP tree, octree, uniform subdivision method and the ABV hierarchy methods. This has resulted in optimal or near optimal performance in all the test cases. In cases where the prediction is off the actual optimal point for subdivision termination, the difference in performance is usually quite small. Thus a major unknown factor affecting the performance of such techniques has been eliminated, allowing them to tune themselves for optimal performance with high confidence.

In the second application, the model has been shown to be only weakly useful in selecting a search structure to be used for a given input scene. In its present form, it is not very useful in predicting the computational requirements for rendering a given input scene. Experimental results on a set of test scenes demonstrate the model is useful in saving computation costs although it does not choose the correct search structure on all the test cases.

Chapter 6

CHARACTERISTICS OF SEARCH STRUCTURES: AN EXPERIMENTAL STUDY

We will next take a close look at some of the properties of hierarchical search structures being used in ray tracing. We are particularly interested in how well these data structures can adapt themselves to scene characteristics. We are also interested in getting an in-depth knowledge of the strengths and weaknesses of each structure. Some of the characteristics that are important to search structures (which are studied here) include the location and orientation of space partitioning planes, the partitioning dimension, effects of bounding volumes, presence of void areas in hierarchies and hierarchy traversal methods. The most commonly used search structures are then described in the context of these characteristics. This has enabled us to get an insight into the characteristics exploited (or not exploited) by each search structure and help us relate performance to scene characteristics.

In this chapter, we present an experimental study of some of the critical properties of search structures being used in ray tracing. We demonstrate how some of these characteristics can be combined to good advantage. This has resulted in the development of a new search structure that is based on k-d trees, which are multidimensional binary structures introduced by Bentley [2][3]. We will describe in detail the construction and use of this new search structure. The superior performance of this structure will be validated with

the cost model developed in chapter 4. We will also show that the hierarchy termination problem that plagues current search structures is solved by this search structure's greater flexibility in its construction algorithm and as such does not require any special treatment, unlike other search structures.

6.1 The Test-bed

We begin by describing the test-bed upon which all experiments were performed to investigate the different characteristics. We have implemented a general space partitioning hierarchical structure. In this system, the partitioning planes are axis-aligned. They can be located anywhere within the scene extents on any of the three dimensions. Bounding volumes, which are rectangular parallelepipeds, can be placed at any node of the hierarchy, enclosing the set of primitives under that node. Both the BSP/octree traversal and the *k-d* tree traversal can be used with this system. The *k-d* tree traversal will be described in a later section. This will be one of the characteristics that will be investigated.

All experiments were conducted on a Sun 4 (SPARC) workstation running 4.0.3 UNIX[1]. Six different scenes were used as test cases, all of which are standard benchmarks (except for the DNA model) available in the public domain [20]. Ray statistics of these benchmarks are described in Table 6.1. All the images were computed at 512 by 512 resolution. Images are illustrated in the color plates of chapter 9. Run times are reported in minutes of cpu time.

[1]UNIX is a trademark of AT&T Bell Laboratories.

Ray counts are in thousands

Scene	*Tetra*	*DNA*	*Arches*	*spd.balls*	*spd.tree*	*spd.mount*
Objects	*1024(P)*	*410(S)*	*4818(P)*	*7382(PS)*	*8191(PSC)*	*8196(PS)*
Lights	*1*	*1*	*2*	*3*	*7*	*1*
Total rays	*299*	*445*	*437*	*1819*	*1676*	*1393*
Visual rays	*262*	*323*	*317*	*722*	*452*	*977*
Shadow rays	*37*	*122*	*120*	*1097*	*1224*	*416*
Hit rays	*44*	*75*	*73*	*873*	*242*	*683*

Table 6.1: Test Scenes

6.2 Characteristics of Search Structures

6.2.1 Space Partitioning

In space partitioning structures, object space is divided up into a collection of convex regions using axis-aligned planes. The search for the closest intersection usually involves testing the partitioning planes for an intersection. The process is recursive, i.e., the regions are recursively partitioned so long as they satisfy the termination criteria used. An example is shown in in Fig. 6.1, with two levels of partitioning.

There are important advantages to using a space-partitioning structure.

1. It enables the use of an ordered search for the closest ray-object intersection. The preferred order is along the path of the ray, so that the search can be terminated as soon as an intersection is found.

2. The partitioning planes help identify the regions visited by each ray. This is a small fraction of the total number of regions in the search structure. Thus large sections of the scene remote from the ray are never examined. In Fig. 6.1, ray $r_1(t)$ examines region 1, while $r_2(t)$ examines regions 1, 4 and 3. With increased partitioning, the number of regions actually examined will be a small fraction of the total.

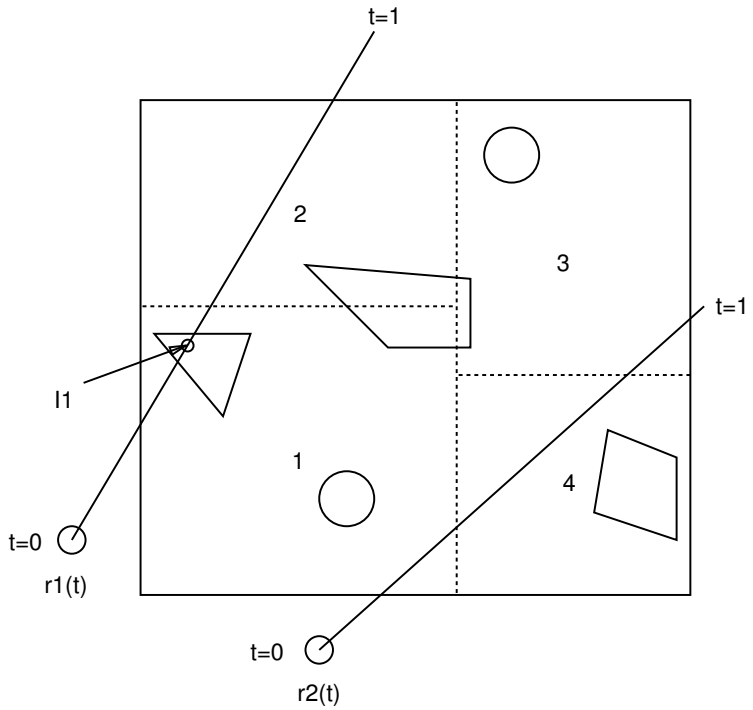


Figure 6.1: Sample Rays

3. The structure is direction independent. With the help of the partitioning planes, an ordering can be determined for a ray with arbitrary origin and direction. Thus all kinds of spawned rays like shadow rays and reflection rays can be traced with no change in the search structure. In Fig. 6.1, ray $r_1(t)$ finds an intersection I_1 and terminates, while $r_2(t)$ examines regions 1, 4 and 3 and terminates without finding an intersection.

4. The partitioning is adaptive, i.e. only parts of the scene that contain large collections of primitives are subdivided. This is important when the object distribution is not uniform. If space was subdivided uniformly, then there would be large collections of empty voxels, and traversal costs would become prohibitively high.

6.2.2 Location and Orientation of Partitioning Planes

In space partitioning structures, the location and orientation of partitioning planes can influence performance. Hierarchical structures like the octree and the Kaplan-BSP tree use axis-aligned partitioning planes which bisect the node extents in each dimension at all nodes of the hierarchy. The node extent is simply the extents of the region represented by the node on all three dimensions. In the octree all three dimensions are partitioned simultaneously, while in the Kaplan-BSP structure the partitioning dimension cycles through the X,Y and Z axes. For further details and differences between these structures, refer to Sections 3.2.2 and 3.2.3.

While it is less expensive to test an axis-aligned plane for an intersection with a ray, this restriction causes it to cross a large number of object primitives. Object primitives that cross a partitioning plane must considered to be on both sides of the plane, since an intersection with this primitive could

be either side of the plane. Restricting the plane locations to be at the center of the node extents has the advantage that it is inexpensive to determine the plane location and thus contributes towards a faster hierarchy construction algorithm. However, it also assumes that objects are uniformly distributed. If they are not, this constraint has the unfortunate effect of creating a large number of empty regions (voxels). This means traversing empty regions, an expense to be avoided, since no intersection can be found in these regions. But the partitioning is adaptive in two respects; a region is not partitioned if it contains less than a small number of objects (which has to be decided in advance), thus avoiding the creation of large numbers of regions containing few or no object primitives, and regions smaller than a preset size are not subdivided any further. The size (for instance, the bounding volume surface area) of the region determines the number of rays that will penetrate it. The smaller it is, the fewer rays that will penetrate the region. Thus, it is not very beneficial to create extremely small regions in the hierarchy.

To sum things up, the creation of empty voxels is itself a source of inefficiency in the first place since no ray-object intersection can be found in such voxels, but the fact that they are not subdivided any further is a desirable factor since it helps the structure adapt itself to the locations and densities of the primitives in the input scene.

Now, let us look at experimental results of using this search structure on the test scenes. In our test-bed, planes are centrally located at each node of the hierarchy. The implementation is very close to the Kaplan-BSP tree. Since the subdivision is identical to the octree, the results should carry over to the latter. Table 6.2 shows statistics for all the scenes. The important figures in this table are the average height of the hierarchy, the total void space in the

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	11.45	43388.03	153.20	2.90
DNA	11.70	32552.10	240.86	6.67
Arches	14.41	59.12	428.22	9.60
spd.balls	20.07	3034.28	547.93	69.93
spd.tree	24.52	48611.76	484.42	73.22
spd.mount	19.53	344.31	273.61	31.58

Table 6.2: BSP subdivision, no bounding volumes, k-d traversal.

hierarchy and of course, the run times. The void space represents a measure of three dimensional space that contain no useful information. Either the volume of the space or the surface area of the bounding volume that encloses this space can be used for measuring void space. We choose to use the area measure. Our aim is to reduce the void space in a hierarchy so that rays that penetrate these spaces can be pruned from further processing.

To compute the void area in the hierarchy, the area of a node in the hierarchy is first determined by computing a bounding volume that encloses all its children in the hierarchy. The surface area of this bounding volume is then calculated. Once a partitioning plane subdivides this node, we have two sub-nodes. The tightest bounding volume enclosing all the primitives in each sub-node is computed, clipped to the bounding volume (or the region extent, if no bounding volume is stored at the node) of the original node and to the partitioning plane. The void area is the difference between the area of the original node and the sum of the areas of its two children. This computation is carried out at each node that is subdivided and summed up to determine the total void area.

The average height in Table 6.2 is rather high, especially for the larger scene models, well beyond the logarithm of the total number of primitives in

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	8.78	105095.42	200.33	3.27
DNA	11.14	60220.91	268.71	7.21
Arches	13.56	252.38	894.52	20.35
spd.balls	14.97	43655.13	708.05	84.39
spd.tree	14.02	716925.55	1097.19	108.88
spd.mount	13.97	1617.71	600.70	43.38

Table 6.3: median-cut subdivision, no bounding volumes, k-d traversal.

the environment. This is important, since all primitives in the hierarchy are stored at the leaf nodes, and reaching them becomes more expensive with increase in the average height. So the first modification we will make is to relax the restriction that partitioning planes have to be located at the centers of the node extents. Instead, in an effort to reduce the average height of the hierarchy, we will place them so as to balance the number of primitives on either side of the partition. The partitioning is thus, along the median of the object primitives. We mentioned earlier the problem of planes crossing large numbers of primitives, something that was unavoidable because there was no other choice for the plane location. Since this is no longer the case, we can take care of this problem by not only trying to obtain a more balanced partition but also minimize crossing primitives. The following figure of merit (fom) accomplishes this.

$$fom = l_{cnt} - r_{cnt} + shr_{cnt} \tag{6.1}$$

where l_{cnt}, r_{cnt} and shr_{cnt} are object counts to the left, right and across the partitioning plane. We want to minimize fom.

Results of using this strategy to locate the plane are shown in Table 6.3. They are not very encouraging. Although the average height of the hierarchy has improved, the new plane locating strategy performs consistently

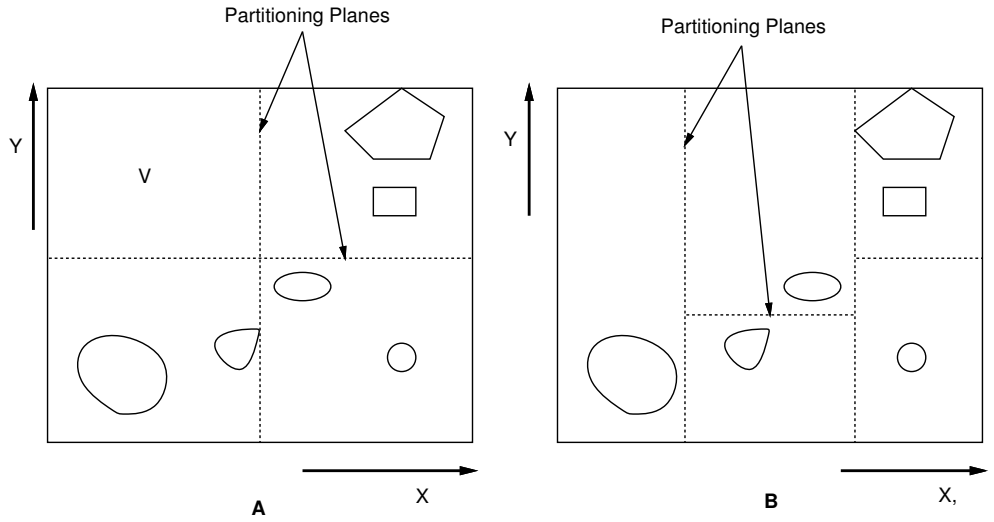


Figure 6.2: Kaplan-BSP vs. median-cut subdivision

worse than the Kaplan-BSP strategy. But a simple explanation is provided by the void area figures in the two tables. By trying to make the locations of the planes more flexible and closer to regions containing object primitives, we have opened up large void spaces.

Consider the simple scene in Fig. 6.2. In 6.2a, the Kaplan-BSP subdivision with two planes creates the void space marked V. This space is not subdivided any further since it does not contain any object primitives. In 6.2b, the median-cut approach tries to balance the primitives across the partition. Thus the void space that appeared in the BSP subdivision will never be isolated by the median-cut scheme because planes are always located in the vicinity of the objects. While this is advantageous in cutting down the cost of reaching the leaf nodes during hierarchy traversal, it allows large collections of the rays to penetrate these void spaces without intersecting any objects, thus nullifying the benefits obtained by the smaller average height of the hierarchy.

One other approach proposed by Macdonald and Booth [32] tries to

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	8.72	99964.82	195.02	3.20
DNA	12.47	31183.64	243.20	6.26
Arches	14.60	153.63	858.33	15.29
spd.balls	15.57	568.55	296.89	33.80
spd.tree	15.20	2243.42	162.93	18.17
spd.mount	14.01	930.41	428.87	30.65

Table 6.4: SA subdivision, no bounding volumes, k-d traversal.

strike a balance between the two strategies considered so far. They try to locate the plane between the spatial median (middle of node extents) and the object median (balanced partition of object primitives). Let b be a partitioning plane of a hierarchy. $SA_l(b)$ and $SA_r(b)$ represent the surface areas on the left and right sides of the hierarchy. n_l and n_r are the object counts on the two sides of the hierarchy. The function to minimize is given by

$$f(b) = SA_l(b).n_l + SA_r(b).n_r \tag{6.2}$$

A good feature of this heuristic is that it takes into account the size of objects. The idea is to strike a balance between surface area and the number of objects that make up that area on each side of the plane. However, this heuristic also fails to address the problem of void area. Once a partitioning plane is located at a node, both sides of the plane can potentially contain large void areas. These areas must be pruned from the hierarchy.

Table 6.4 shows the results of using this plane choosing approach. The void areas are somewhat lower than the median-cut strategy, but still much higher than the Kaplan-BSP strategy. Although performance improves on two of the scenes, on the rest they are nearly the same or worse.

6.2.3 Role of Bounding Volumes

There are important advantages to using bounding volumes to optimize ray tracing.

1. Around object primitives, bounding volumes provide a simple and inexpensive non-intersection test. Only if the ray penetrates the bounding volume does the primitive inside it needs to be tested for intersection.

2. By surrounding a collection of objects completely with a bounding volume, large sections of object space can be pruned away, drastically reducing the ray search space. This is because if a ray misses this bounding volume, the entire collection of objects inside need not be examined at all. Bounding volumes are more effective than space partitioning structures in this respect, since, in general, they provide tighter enclosures around object collections. Also, a bounding volume can be made arbitrarily tight [26], although the cost of testing the bounding volume for an intersection also increases, as we saw in Section 3.3.2. Grouping objects into clusters of increasing size and surrounding them with bounding volumes results in a bounding volume hierarchy. Any node in this hierarchy represents a cluster of objects contained in its subtree.

3. The algorithms used to construct bounding volumes are critical to performance. The best known hierarchy to date is the ABV hierarchy, which was described in Section 3.3.1. Its biggest advantage is the automatic construction of the hierarchy, which facilitates rendering complex environments. It also makes it the fastest method for constructing a bounding volume hierarchy.

Scene	Time(min.)
Tetra	4.14
DNA	11.90
Arches	14.16
spd.balls	55.33

Table 6.5: ABV Hierarchy Performance.

Table 6.5 shows results of using the ABV hierarchy on the test scenes. In general, its performance is worse than the BSP subdivision but better than the median-cut strategy. The major problem in bounding volume hierarchies is that while void space is not a problem (the bounding volumes in the hierarchy cull most of it out) the ray intersection search is not performed along the path of the ray. All bounding volumes in the hierarchy that that are in the path of a ray have to be queried for an intersection before the closest intersection can be reported.

6.2.4 Adding Bounding Volumes to Space Partitioning Structures

Space partitioning hierarchies optimize ray tracing by providing an ordering of regions visited by the ray independent of the ray origin or direction. This ordering also makes it easy to terminate the search once the closest intersection is determined. Since only regions visited by the ray are examined for intersection, the structure helps avoid examining regions that are far from the path of the ray. Space partitioning structures are adaptive, concentrating the partitioning in the vicinity of objects. Since all the partitioning planes are axis-aligned, they have the potential to create large void spaces which are a source of inefficiency. Bounding volume hierarchies, on the other hand, optimize ray tracing by culling away large sections of object space and provide a compact representation for the objects at every node of the hierarchy. They

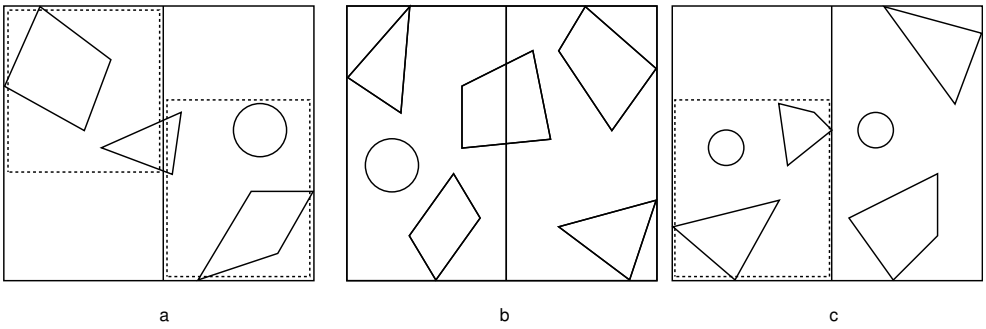


Figure 6.3: Bounding Volumes

are more effective than space partitioning hierarchies in this because of the greater freedom available in choosing the location, orientation and shape of the bounding volume. Also, they can be made to fit as tight as possible around object clusters, thus reducing the ray search space even further. Rays that do not intersect a bounding volume in the hierarchy will not need its subtree (containing bounding volumes and/or object primitives) to be examined any further. Lastly, unlike space partitioning hierarchies, the bounding volume hierarchy traversal is not along the path of the ray, although a partial ordering can be obtained by sorting and processing bounding volume intersections in sorted order [26].

Current search structures have not exploited the advantages of using bounding volumes in space partitioning structures. Yet, this seems to be the most natural thing to do, since the advantage of one overcomes the disadvantage of the other. The problem of void areas in space partitioning structures can be overcome by enclosing objects at nodes of the hierarchy with bounding volumes. Bounding volume hierarchies are inefficient because rays are not traced along their path and hence do unnecessary intersection calculations. Although adding bounding volumes to space partitioning structures does introduce additional bounding volume tests, we will show that their benefits in culling void space

will easily outweigh the additional cost.

To avoid unnecessary bounding volume tests, bounding volumes should not be used at every node of a space partitioning structure. Refer to Fig. 6.3. In 6.3a, bounding volumes are required on both sides of the partition (dotted lines indicate bounding volumes, in 6.3b, no bounding volumes are required and in 6.3c, only the left side needs it. Thus bounding volumes should be used only where they result in culling a large amount of void space.

This is then our next strategy. We will add bounding volumes to space partitioning structures where they will result in cutting down void space. Tables 6.6, 6.7 and 6.8 show results of the search structures using the three plane choosing strategies with bounding volumes added. The performance improves for all three techniques. The performance improvement is the most dramatic for the median-cut scheme. The improvement in performance is validated by the large decrease in void areas (brought forth by the bounding volumes, understandably). The performance of the SA scheme is better than the median-cut scheme. However, these timings do not include the search structure construction times. It must be remembered that the plane choosing strategy using the SA heuristic is more expensive. In addition to the binary search that needs to be performed to determine the object median, a linear search between the spatial and object medians is required in order to minimize the function in Equation 6.2. Including the construction times, Table 6.9 illustrate the timing results on all the test cases. It is seen that the SA method is still better than the median-cut scheme three out of five times. This is not surprising since the median-cut subdivision strategy is a special case of the SA heuristic. The SA heuristic in addition to considering object counts on both sides tries to minimize the surface area (of the bounding volumes of the two object sets in our

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	11.47	0.07	144.45	2.48
DNA	11.91	723.45	229.10	6.26
Arches	15.79	0.95	304.17	8.17
spd.balls	15.74	1.95	260.74	33.32
spd.tree	13.77	61.26	155.97	21.40
spd.mount	13.91	5.09	272.00	25.98

Table 6.6: BSP subdivision, bounding volumes, k-d traversal.

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	8.78	960.63	152.55	2.56
DNA	11.14	1363.29	216.40	5.87
Arches	13.72	1.442	320.21	8.02
spd.balls	14.97	4.47	259.53	30.42
spd.tree	14.02	6.77	125.15	13.50
spd.mount	13.97	1617.71	282.45	24.60

Table 6.7: median-cut subdivision, bounding volumes, k-d traversal.

implementation) which directly corresponds to reducing void space.

6.2.5 Traversal Methods

We next take a look at two different traversal methods that can be used in space partitioning hierarchies. The BSP/Octree traversal has already been described (Section 3.2.2). The second traversal method that we have developed will be called the k-d tree traversal. This will now be described.

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	8.72	735.80	144.84	2.23
DNA	10.98	866.59	212.11	5.58
Arches	14.60	1.00	286.01	7.08
spd.balls	15.20	3.99	244.51	26.71
spd.tree	14.51	5.52	127.55	12.92
spd.mount	14.40	4.63	244.63	19.36

Table 6.8: SA subdivision, bounding volumes, k-d traversal.

Scene	Run time	
	k-d	SA
Tetra	2.6	2.45
DNA	5.93	6.37
Arches	9.11	8.75
spd.balls	31.02	28.05
spd.tree	14.31	14.87
spd.mount	25.35	20.74

Table 6.9: Total run times: *k-d* versus SA

6.2.5.1 The *k-d* Tree Traversal Algorithm

Given a ray and the root to a space partitioning hierarchy, the first step is to compute an intersection with the bounding volume of the objects under this node (if there is no bounding volume at the node, the node extent can be used). If there is an intersection, then we need to determine if this node is a leaf or an internal node of the hierarchy. If it is a leaf node, then all the objects in its list will be tested for intersection, and the closest of these (if any) will be reported. If primitive objects contain bounding volumes, then the primitive is tested for intersection only if the ray penetrates the bounding volume. On the other hand, if this is an internal node, we have two different cases:

1. The ray lies entirely on one side of the partitioning plane.

2. The ray crosses the plane.

For case 1 the region containing the ray segment (the ray has now been clipped to the bounding volume of this node) needs to be searched. For case 2, the intersection point between the ray and the partitioning plane is determined. The ray is then split into two parts. Both regions could possibly be examined, but the region close is closer to the ray origin is examined first. If

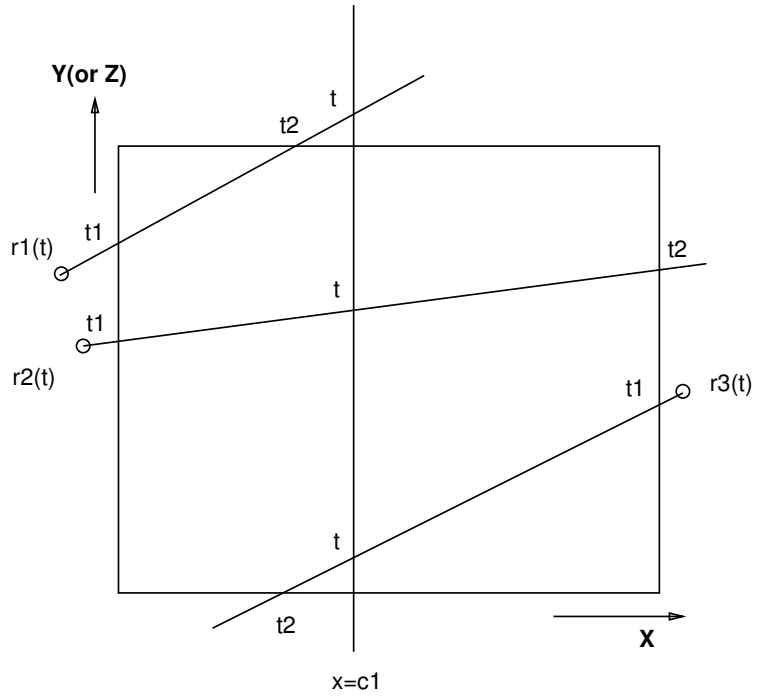


Figure 6.4: Ray Traversal.

an object intersection is found in this region, then we are done, else the second region has to be examined. This process is recursively applied until we find a ray-object intersection or run out of regions (and hence, objects).

Some examples are shown in Fig. 6.4. Here $r1(t)$, $r2(t)$ and $r3(t)$ are three different rays. $t1$ and $t2$ are the parametric intersection points with the region, and t, the intersection with the partitioning plane $x = c_1$. The top ray visits only one of the 2 regions, which is recognized by the fact that $t1 < t2 < t$. The actual region visited by the ray is determined by the x component of the ray direction. For the top ray, the direction along the x axis is positive, i.e. x increases along the path of the ray, thus identifying region 1. Otherwise, region 2 will be visited. The middle and bottom rays visit both regions, because $t1 < t < t2$. The x component of the ray direction determines the traversal order. For the middle ray, the direction is positive, so the order

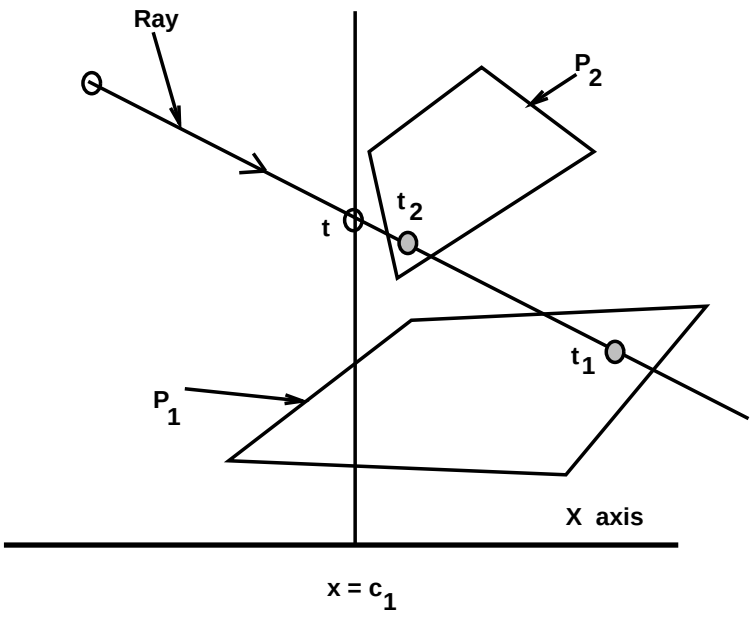


Figure 6.5: A Case 2 Possibility

is region 1 followed by 2, while it is 2 followed by 1 for the bottom ray because its x direction is negative. A similar strategy holds for identifying the traversal order when the partitioning dimension is y or z.

Case 2 has a complicating factor. An intersection from the first of the two regions could be in the second region, as the intersected object could be part of several regions. Before claiming this as the closest intersection point, we must make sure that the t value at this intersection is within the t interval defined by the ray segment end points. Fig. 6.5 illustrates such a case with a partitioning plane $x = c_1$.

P_1 and P_2 are two polygons. P_1 is part of both regions whereas P_2 is entirely to the right of the partitioning plane $x = c_1$. P_1 will be examined first, since it is partly on the side containing the ray origin. But the closer intersection is at t_2 with P_2. The intersection t_1 with P_1 will be greater than t, the intersection with $x = c_1$. This will result in examining the the region

$x > c_1$ and hence, P_2.

It is possible for an object primitive to occupy several adjoining regions. To avoid examining an object primitive more than once, ray signatures are maintained in all the object records. Each object is examined for intersection only if its signature is different from that of the current ray. If not, it is tested for intersection and the ray signature is written onto its record.

Note that this traversal algorithm differs significantly from that of Kaplan-BSP tree traversal. The traversal algorithm is described in Algorithm 6.1.

6.2.5.2 Comparing the Two Traversal Methods

The regions identified by both traversal methods in tracing any ray are identical. The difference lies in how this is accomplished. Let us look more closely at the operations performed by the two methods (the numbers given in Tables 6.10 and 6.11 pertain to our implementation and can be replaced by equivalent quantities for other implementations).

Tables 6.10 and 6.11 show the operations performed in both of these traversal methods. h_{avg} is the average height of the hierarchy. For the BSP/Octree traversal, the total operations is an average count while in the k-d tree traversal, the total operations is a worst case count. For the k-d tree traversal, very often, neighboring regions will be close to each other in the hierarchy. Thus, if the next region to be processed is close to the region just processed in the hierarchy, then both regions will have a common ancestor that is relatively deep in the hierarchy. Hence the number of plane intersections and branching decisions done to get to the new region will be much smaller than

```
Input: root of the k-d tree, ray end points.
Output: true/false from the function and if true the intersection.

traverse (root,ray,tstart,tend,inters)
{
      if (inters_volume (root→vol))
      {
            if (root→type == LEAF)
                  return (list_inters (root→list,tstart,tend,inters))
            else
            {
                  rayclip (root→vol, &t_start, &t_end)
                  path = ray_path (root,ray,tstart,tend,&tmid)
                  switch (path)
                  {
                        LR :   if (traverse (root→left,ray,tstart,tmid,inters))
                                     return (TRUE)
                               return (traverse (root→right,ray,tmid,tend, inters))
                        RL :   if (traverse (root→right,ray,tmid,tend, inters))
                                     return (TRUE)
                               return (traverse (root→left,ray,tmid,tend, inters))
                        L :    return (traverse (root→left,ray,tstart,tend, inters))
                        R :    return (traverse (root→right,ray,tstart,tend, inters))
                  }
            }
      }
      return (FALSE)
}
```

Algorithm 6.1: *k-d* Tree traversal.

h_{avg}.

Looking at the expressions for the two methods, it is not clear which method is better. The BSP/octree traversal does more work in moving from voxel to voxel while the *k-d* tree traversal is doing more work reaching the leaf nodes. For very dense scenes, if an intersection is found within the first few regions the ray encounters, then the BSP/octree method does less work. This is because the *k-d* tree method determines the order (in advance) for regions that may never be visited by the ray because of early termination. If the

Operation	flops
Reaching leaf node	h_{avg}
Determining face through which ray exits	15
Computing exit point	6
Extending ray into the next region	9
Total cost/region examined	$30 + h_{avg}$

Table 6.10: BSP/octree Traversal Operations

Operation	flops
Plane intersection at each node	2
Branching decision at each node	1.5
Total cost to get to leaf node	$3.5h_{avg}$

Table 6.11: K-d Traversal Operations

ray travels a number of regions and if there are locality properties between successive regions in the hierarchy, then the *k-d* tree method will do better because all its computation is done in ray parametric space and no coordinates are computed. One other restriction for the BSP/octree traversal is that all the regions visited must be contiguous, since the ray is 'extended' from region to region. The amount by which the ray is extended is determined by the smallest voxel in the hierarchy. If there were gaps between voxels, this traversal could put a ray origin in a gap, and there would be no corresponding voxel containing the origin. The *k-d* tree traversal has no such restriction.

Let us now look at experimental results obtained by using the two traversal methods on the same set of test cases. Table 6.12 shows the results of using the Kaplan-BSP traversal on the test cases. When compared to Table 6.2, which uses the *k-d* tree traversal, it is clear that the *k-d* tree traversal is superior in practice.

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	11.46	433388.03	249.42	3.46
DNA	11.71	32552.10	339.36	7.62
Arches	15.37	59.12	593.98	12.05
spd.balls	20.06	3034.28	1053.52	102.45
spd.tree	24.52	48611.76	1146.51	108.58

Table 6.12: BSP Subdivision, no bounding volumes, BSP Traversal.

6.3 A New Search Structure for Efficient Ray Tracing Based on k-d Trees

Our experimental study has pointed us towards several new strategies for building a new search structure for efficient ray tracing.

1. The surface area (SA) heuristic defined by Equation 6.2 is to used to locate the partitioning plane at every node of the hierarchy.

2. Bounding volumes are used at nodes of the hierarchy where they result in cutting down void space (we have already explained how to compute a measure of the void space at a node).

3. The *k-d* tree traversal method must be used to trace rays through the search structure.

A search structure with the above features falls under the classifications of *k-d* trees, which are multidimensional binary search structures, originally proposed by Bentley [2][3] for applications in range searching.

Henceforth, when we refer to the *k-d* tree, it is to be understood that we are referring to the search structure (with characteristics described above) that is being used for ray tracing.

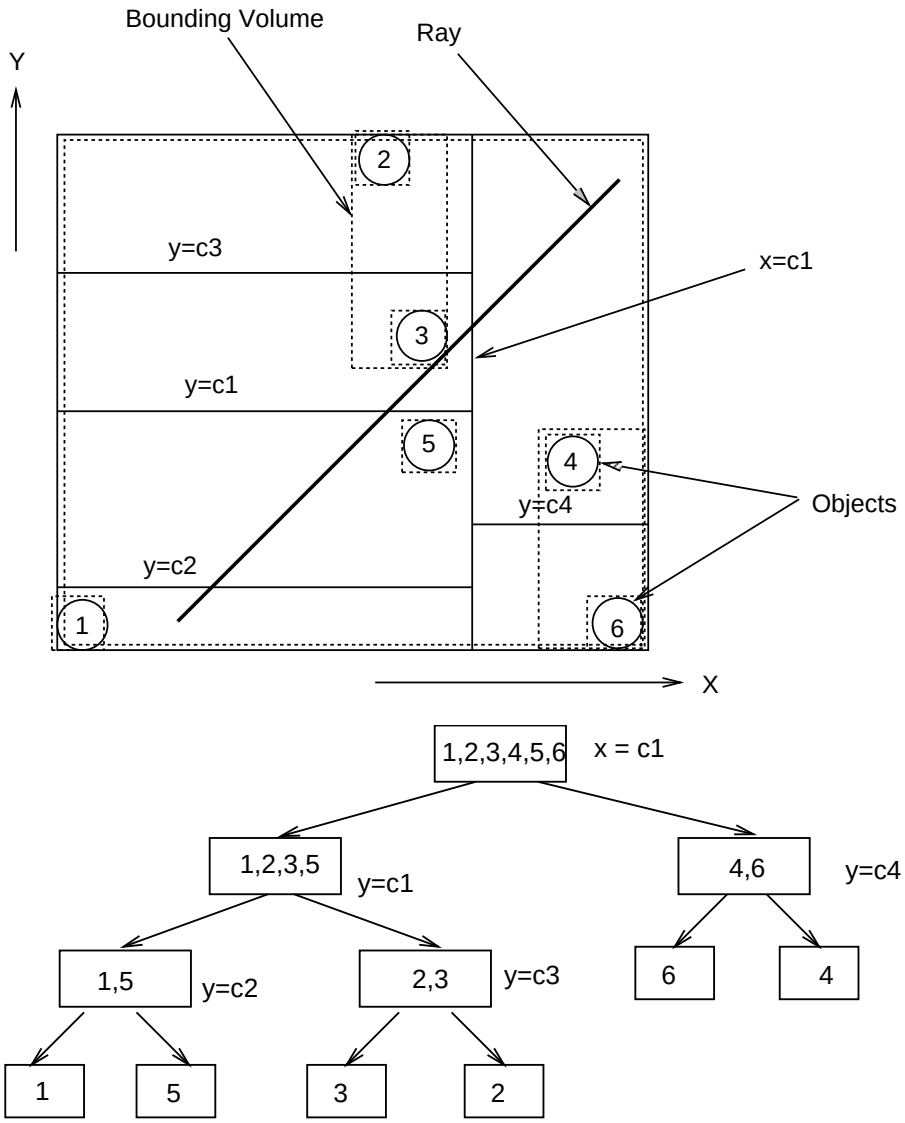


Figure 6.6: A Sample *k-d* Tree

A 2D example of a k-d tree subdivision and the corresponding tree is shown in Fig. 6.6.

We have already described the traversal of the k-d tree. We will now describe the construction of the k-d tree.

6.3.1 Construction of the k-d Tree

There are four parts to this preprocessing step:

- Determining the object median.
- Determining the spatial median.
- Determining the plane that minimizes the function in Equation 6.2.
- Partitioning the object set across the chosen plane.

6.3.1.1 Determining the Object Median

This process involves a binary search along the three coordinate axes within the scene extent. Algorithm 6.2 illustrates the procedure. It consists of the following steps:

1. Choose the plane midway between the scene maximum and minimum along the particular coordinate axis.
2. Classify each object primitive of the scene according to its position on the left, right or both sides of the chosen plane. To determine how good this partition is, compute a figure of merit as defined by Equation 6.1 If this is unsatisfactory (determined by the termination conditions in step 4), we will need to move the plane to the region that contains the larger

number of objects. Whichever region is chosen, *only the objects in that region need to be partitioned* for the new plane choice. To make this clearer, refer to Fig. 6.7.

Here the plane $x = c_1$ partitions the object set into *l_cnt, r_cnt and shr_cnt*. Plane $x = c_2$ is the new plane choice since $l\_cnt > r\_cnt$. Only objects to the left of $x = c_1$ need to be partitioned with respect to $x = c_2$ since objects to the right of $x = c_1$ form a subset of objects to the right of $x = c_2$. A similar argument applies when $l_{cnt} < r_{cnt}$.

3. To choose the new plane as in step 1, we need to update the interval bounds. If the new plane is to be located to the left of the present plane, then the latter becomes the right bound; if it is to the right, then the newly chosen plane becomes the left bound. In Fig. 6.7, $x = c_2$ is to the left of $x = c_1$ and so $x = c_1$ is the new right bound.

4. If the figure of merit computed above is less than a threshold, then we are done. Else we must determine whether we still need to continue the binary search in this coordinate direction. As illustrated in Algorithm 6.2, there are three conditions:

   - The left and right bounds differ by less than a chosen resolution.
   - All objects are shared between the two regions (i.e. no suitable plane can be found for subdivision). In this case the figure of merit will equal the original number of objects.
   - The number of objects in both regions are equal. This will not necessarily give us the best partition in this dimension since the number of objects crossing this plane could be large. As we do not know the side of the plane in which to continue the search, we

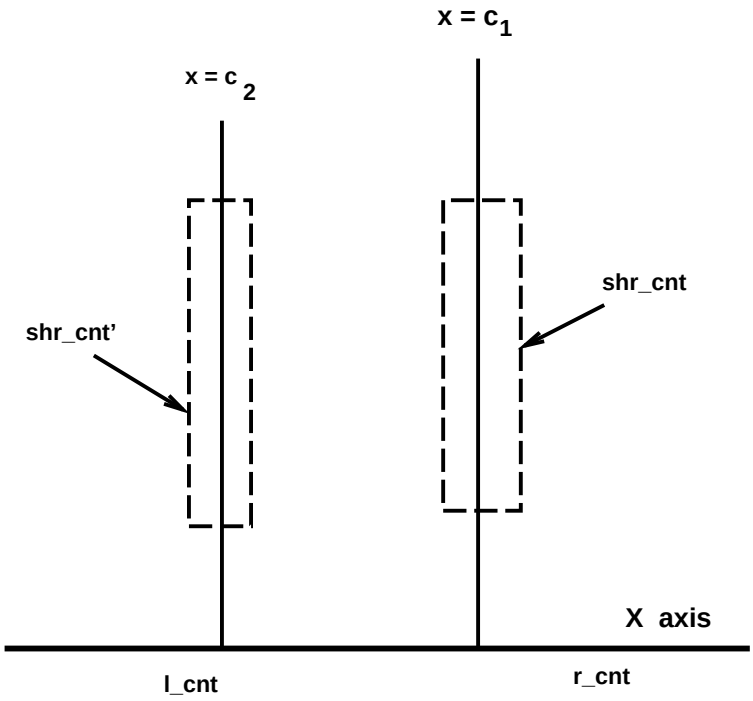


Figure 6.7: Object Classification

stop the search in this dimension and continue with the remaining dimensions.

6.3.1.2 Determining the Spatial Median

This is the plane that partitions the region or space under consideration into two equal sized subspaces. If no bounding volumes are being used in the search structure to enclose collections of objects, then the plane that is located midway between the extent bounds for the partitioning dimension is the spatial median. If bounding volumes are being used in the structure, then this plane does not necessarily partition the original space into two equal subspaces. In Fig. 6.8A the spatial median is at the center of the extent. The same region shown with bounding volumes in 6.8B does not partition the space into two

Input: list of objects, number of objects, scene volume.
Output: partitioning plane, divisible flag.

```
choose_plane (object_list, object_cnt, vol, plane, divisible)
{
  i = X
  fom = best_fom = +∞
  while (fom > threshold && i ϵ (X,Y,Z))
  {
      choice = (vol_imin + vol_imax)/2
      classify (obj_list, choice, &new_side, &fom)
      if (new_side == LEFT)
        vol_imax = choice
      else (if new_side == RIGHT)
        vol_imin = choice
      else i++
      if (fom < best_fom)
      {
        best_fom = fom
        plane→value = choice
        plane→indx = i
      }
      if (|vol_imin − vol_imax| < Resolution_i)
        or (fom == object_cnt)
        i++
  }
  *divisible = (fom == object_cnt)
}
```

Algorithm 6.2: Choosing a Partitioning Plane.

equal sized regions. The size of the two subspaces is determined by the sizes of the bounding volumes, which, in turn, is determined by the objects they enclose.

In the presence of bounding volumes, the spatial median can be determined by a binary search, in a manner similar to the search for the object median. In this case, the goal is to determined the plane that best balances the surface areas of the bounding volumes on both sides of the partitioning plane. The partitioning plane is successively located in the region (or half-space) that

```
Input: object list, object_cnt, scene volume.
Output: root of the k-d tree.

TREE_PTR build_k-d (object_list, object_cnt, volume)
{
  choose_plane (obj_list, object_cnt, volume, &plane, &divisible)
  if (divisible)
  {
    create 'root node'
    classify (obj_list, plane, &l_list,&r_list &l_cnt, &r_cnt)
    if (left_cnt > threshold)
      root→left = build_k-d (l_list, l_cnt, getvol(l_list))
    if (root→left == NULL)
      root→left = obj_list
    if (right_cnt > threshold)
      root→right = build_k-d (r_list, r_cnt, getvol(r_list))
    if (root→right == NULL)
      root→right = obj_list
    return (root)
  }
  return (NULL)
}
```

Algorithm 6.3: Building the *k-d* Tree.

has the larger bounding volume surface area, until the areas are very close to each other. The figure of merit for this search is simply the absolute value of the difference between the two bounding volume surface areas. Minimizing this will balance the surface areas.

6.3.1.3 Determining the Partitioning Plane

A linear search is performed between the object and spatial medians. For each plane choice, Equation 6.2 is evaluated This is compared against the currently store minimum value. A record of the plane with the minimum function value is maintained. This will be the chosen plane at the end of the search.

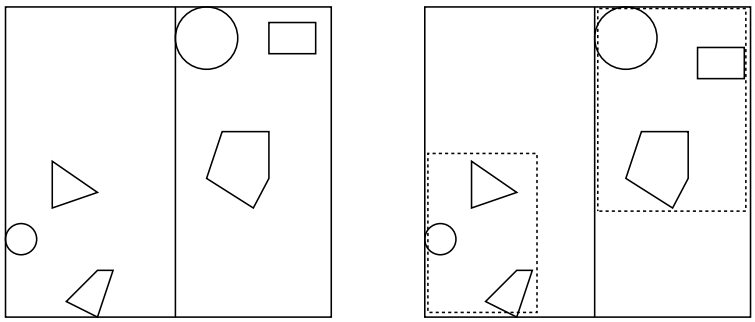


Figure 6.8: Determining Spatial Median.

6.3.1.4 Building the k-d Tree

Once a partitioning plane has been determined, the object primitives are partitioned into two subsets. Objects that cross the plane are counted in both regions (which can be maintained by pointers to the object record). The partitioned subsets are recursively partitioned until the termination criteria are satisfied. The procedure is outlined in Algorithm 6.3.

6.3.2 Validating the Results of the k-d Tree Using the Cost model

With the desirable characteristics incorporated into our new search structure, we will next like to validate its superiority over current search structures. The first result that we would like to demonstrate is the automatic termination criteria built into the construction of the *k-d* tree is sufficient for it to operate at the correct subdivision depth for obtaining the best performance. Secondly, we will compute the cost model at this subdivision depth and show that the predicted cost is less than current seach structures.

In the *k-d* tree (using either a median-cut plane choosing strategy or

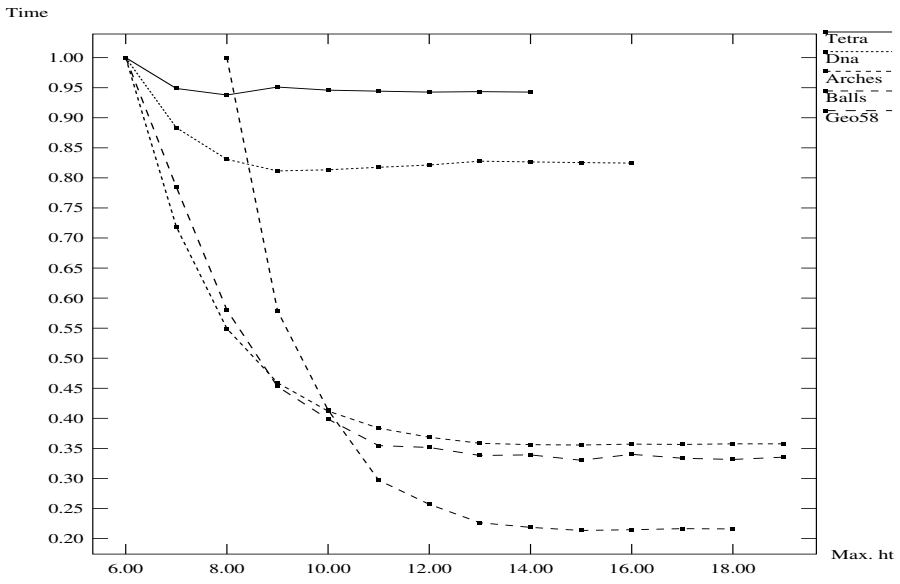


Figure 6.9: K-d Tree Performance Characteristics

the SA heuristic) the flexibility in plane locations allow us to stop subdividing when its not possible to place at least one object on either side of the partitioning plane. Continuing subdivision beyond this point only serves to increase the traversal cost, since the same number of objects will remain at each of the new nodes that are created by this additional subdivision. However, it is possible that stopping subdivision before this condition is reached might be more optimal. We will demonstrate that this is not so by experimental characteristics.

Fig. 6.9 shows the experimental characteristics of the *k-d* tree for three different scenes. It is clear from these characteristics the performance is best at the point where the hierarchy construction would not have subdivided any further. Fig. 6.9 illustrates the *k-d* tree performance characteristics on different scenes for predicting termination criteria. In each case, optimal performance is reached only at the height at which the *k-d* tree would not have subdivided any further. Because of the greater flexibility in the construction in

the *k-d* tree, its construction algorithm is able to set termination criteria in the tree building algorithm. When the plane choosing algorithm determines that it is not possible to place at least one object primitive on one side of the partitioning plane completely, the algorithm terminates subdivision (at any node of the tree). Thus, the model confirms these results. One difference in the model evaluation for the *k-d* tree from the octree and BSP hierarchies is that the cost of processing bounding volumes has to be added to the traversal cost. For this, we need to determine the average number of bounding volumes along any path of the *k-d* tree. During the preprocess, all paths of *k-d* tree have to be examined to determine the average number of bounding volumes per path. This is again a weighted average, since leaf nodes which are smaller in size will be visited less often. This value is multiplied by the cost of a bounding volume test and then added to traversal cost.

The fact that the *k-d* tree stops subdivision after a certain depth is reached is indicated by the curves remaining flat after this point. Although the construction algorithm allows subdivision beyond this point (this will cause the octree and BSP structures to subdivide) the *k-d* tree terminates itself when it realizes that no more benefits can be obtained by further subdivision.

Table 6.13 shows the predicted costs of all the test scenes using the cost model. Comparing this to the figures in Tables 5.6 5.7, 5.8, 5.9 and 5.10, it is seen that in all the cases except the DNA, the model predicts the superior performance of the *k-d* tree. The experimental results *k-d* tree with the SA heuristic have been shown earlier in Table 6.8

Scene	Predicted Cost (float ops./ray)
Tetra	69.8
DNA	90.2
Arches	131.1

Table 6.13: k-d Tree Predictions.

6.4 Conclusions

In this chapter, we have performed an extensive study of some of the important characteristics of search structures currently being used in ray tracing. Their strengths and weaknesses have been analyzed. We have used results from this study in developing the *k-d* tree, a new search structure that combines the advantages of both space-partitioning structures and those based on bounding volumes alone. The superior performance of the *k-d* tree over previously developed search structures has been demonstrated experimentally. These results have been validated by the cost model that we developed and used successfully for other search structures. The *k-d* tree has been shown to be a more flexible search structure with better properties for adapting itself to scene characteristics. It does not suffer from the termination problem common to other search structures. Its flexibility allows it to terminate itself at the point where no additional benefits can be obtained from further subdivision.

Chapter 7

APPLYING SPACE SUBDIVISION TECHNIQUES TO VOLUME RENDERING

In this chapter, we will apply the k-d tree search structure that we proposed in the previous chapter to an important application of computer graphics: visualization of data from scientific applications. We present a new ray-tracing algorithm for volume rendering which is designed to work efficiently when the data of interest is distributed sparsely through the volume. A simple preprocessing step identifies the voxels representing features of interest. Frequently this set of voxels, arbitrarily distributed in three dimensional space, is a small fraction of the original voxel grid. The voxels are then stored in a k-d tree. The tree is then efficiently ray-traced to render the voxel data. The k-d tree is view independent and can be used for animation sequences involving changes in positions of the viewer or positions of lights. We have applied this search structure to render voxel data from MRI, CAT Scan and electron density distributions.

7.1 Introduction

An increasingly important application of computer graphics technology is in providing visualization tools to help scientists in a number of fields understand massive amounts of data. Some of these fields include medical imaging, molecular modeling, computational fluid dynamics, seismology, weather models and oceanography. In most of these applications, the data generated

are usually too large to interpret directly in their raw form. Also, the data models usually contain a large number of features which are difficult to study all at once. A visual representation of these features, either individually or in some reasonable combination, is desirable for a better understanding of the underlying phenomena.

Many of these data-sets are scalar or vector fields of functions sampled in three spatial dimensions. For example, medical imaging data consists of scalar density values at each vertex of a three dimensional grid. These 'voxel' data are either input directly to a rendering program, in which case the visualization procedure is termed 'volume rendering', or converted to an intermediate representation, for example a surface model, before rendering.

The advantage of creating a surface model from a volumetric representation is that it is often a compact encoding of the characteristic that is being visualized. A surface model, built of polygonal primitives, for instance, can be rendered efficiently with special purpose graphics hardware. The model needs to be created only once and can be viewed from any direction. If the number of surface primitives is not too large, rendering can often be performed in real time, a very useful feature in scientific visualization. A popular method for creating surface models from voxel data is the Marching Cubes method [31].

Most often, creating a surface model involves making a binary classification decision of whether a surface passes through a voxel or not. This leads to introduction of artifacts in the image. Also, it may not make sense to create surfaces for certain kinds of volume data. In these cases, an alternate solution such as the direct rendering of volumetric data is preferred. Direct volume rendering techniques are generally based either on a ray-casting approach [41][28][29][30][49] or on the projection and compositing of preprocessed voxels

onto the image plane [49][12]

Ray-casting techniques sample points along the path of each ray (as often as needed) that is cast into the volume. A weighted sum of the contributions of all these points is projected onto the view plane. The images obtained using this method typically have fewer artifacts than surface modeling methods because no binary classifications need to be made, although the point sampling involved can also be a source of errors due to undersampling. Also the grid is sampled at a greater level of detail, thus providing a more accurate picture of the volume data. Similar advantages can be obtained with projection and compositing methods.

Unfortunately, direct volume rendering using ray tracing is generally more time consuming than surface rendering. Such methods, in general, do not take advantage of standard graphics hardware. Though the images produced are of higher quality than those generated from surface models, each image is more expensive to compute. It can be quite expensive to generate animation sequences, which are often a key to understanding scientific phenomena.

Surface modeling approaches to visualizing volumetric data take advantage of the fact that very often the features of interest to a scientist are contained in only a small portion of the original voxel data. By suitably identifying this subset of voxels and representing them as a set of surfaces, considerable savings in computation and space can sometimes be obtained. A surface may be an effective way to visualize data in cases in which a single type of real surface is being identified in the volume data. Frequently, however, either many different surfaces are of interest or the data actually contains no real surfaces. In these situations, direct volume rendering may provide a more useful way for investigators to understand the information of interest in the data.

In this chapter, we present a volume rendering algorithm which takes advantage of the lack of interesting information in a large fraction of voxel data in many applications without representing the interesting voxels as surfaces. Our goal is to achieve the benefits of the relatively fast rendering and compact representations of surface models while retaining the ability to effectively represent data which are poorly suited to surface modeling.

Our strategy is to first identify the voxels which do not contain interesting data and remove them, rather than attempting to identify a set of interesting voxels which can constitute a surface. After this initial step, we are left with clumps of interesting voxels distributed throughout the original volume. We incorporate this data into a *k-d* tree. The construction and use of the *k-d* tree is the same as was done for rendering surface models. For regular grids, axis-aligned planes allow us to construct the tree entirely using integer arithmentic. These and other optimizations due to the special nature of the data that is being rendered will be described later.

The motivation for applying space subdivision techniques to volumetric rendering comes from their success in accelerating the ray tracing of surface models [17][25][39][18][1][13]. Levoy [29] and Meagher [33] have used octrees for rendering volume models. In the previous chapter an extensive investigation of different characteristics of ray tracing hierarchies showed that for surface models, the *k-d* tree is a more adaptive and flexible data structure, principally because it combines the advantages of pure space partitioning structures such as the octree and those based on bounding volumes.

An important advantage of this search structure is its view independence. Animation sequences involving changes in viewer locations or positions of lights require no change in the search structure. Slicing the volume data to

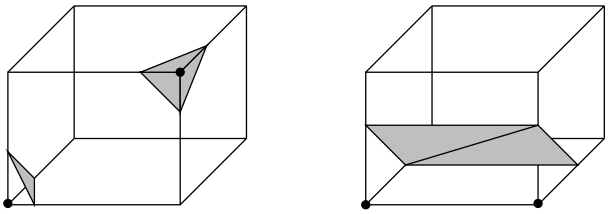


Figure 7.1: Triangulated Cubes.

look at the internals of a feature also can use the same search structure with a minor modification in the preprocessing step. We have used this structure to efficiently render animation sequences of a human heart (from an MRI data-set) and scalar fields of electron density distributions.

The remainder of the chapter is organized as follows. First, we survey some major existing surface rendering and ray-tracing techniques for volume data in order to identify techniques for identifying and processing interesting voxels which we will also make use of. Subsequently, we present a detailed description of our algorithm, and then examine implementation results on the data mentioned above.

7.2 Visualizing Volumetric Data

Among the different methods of visualizing volumetric data, two techniques are commonly used in medical imaging and molecular modeling applications. The first is the Marching Cubes method, which outputs polygonal surfaces of a certain density threshold value. The second method is based on ray tracing and directly samples the voxel grid.

The Marching Cubes method builds triangle models of constant data value surfaces from 3-D scalar fields. A threshold density value (or surface constant) is first selected. This value is compared with the density values at

the eight corners of each voxel. If it falls within the density range of any of the edges of the voxel, then the surface intersects that edge. The intersection points, determined by linear interpolation from the edge densities define one or more planar surfaces. These surfaces are then triangulated. Two examples are shown in Fig. 7.1. Before rendering, a unit normal is computed for each triangle vertex. For a constant data value surface, the gradient vector is normal to the surface. The gradient at each vertex (i, j, k) of the grid is computed using central differences. Let this be $\bigtriangledown f(\vec{x_i})$. It is given by

$$\begin{aligned}
\bigtriangledown f(\vec{x_i}) &= \bigtriangledown f(x_i, y_j, z_k) \\
&= \left\{ \begin{array}{l} \frac{1}{2\Delta x}[f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k)], \\ \\ \frac{1}{2\Delta y}[f(x_i, y_{j+1}, z_k) - f(x_i, y_{j-1}, z_k)], \\ \\ \frac{1}{2\Delta z}[f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1})] \end{array} \right\}
\end{aligned}$$

and the unit normal is given by

$$N(\vec{x_i}) = \frac{\bigtriangledown f(\vec{x_i})}{|\bigtriangledown f(\vec{x_i})|}$$

where

$$\begin{aligned}
f(x_i, y_j, z_k) &= \text{density at location } (x_i, y_j, z_k). \\
\Delta x, \Delta y, \Delta z &= \text{grid spacing along the three} \\
&\quad \text{dimensions.}
\end{aligned}$$

Normals at the vertices of the triangles are calculated by linear interpolation from the corner gradients. Once the normals for the vertices of all the triangles are computed, the triangles can be rendered on any standard graphics workstation.

In the ray tracing approach, rays are cast into the voxel grid, through an imaginary projection plane. Each of these rays is sampled at equal intervals along its path through the grid. This can exploit the use of incremental techniques [6][15]. At each sample location, the voxel density, opacity and local gradient are determined. The voxel density is usually obtained by linear interpolation from the corner density values. The opacity can be a made a function of the density, although this is not necessary. Theoretically, the opacity can be any meaningful function. For instance, if it were a step function peaking at a certain density value, then we end up with iso-surfaces, as in the Marching Cubes method. A method proposed by Levoy [28] to compute opacity is as follows:

$$\alpha(\vec{x_i}) = \alpha_v * \begin{cases} 1 & \textbf{if } |\bigtriangledown f(\vec{x_i})| = 0 \\ & \quad \textbf{and} \\ & f(\vec{x_i}) = f_v \\ 1 - \frac{1}{r}\left|\frac{f_v - f(\vec{x_i})}{|\bigtriangledown f(\vec{x_i})|}\right| & \textbf{if } |\bigtriangledown f(\vec{x_i})| > 0 \\ & \quad \textbf{and} \\ & f(\vec{x_i}) - r|\bigtriangledown f(\vec{x_i})| \\ & \quad \leq f_v \leq \\ & f(\vec{x_i}) + r|\bigtriangledown f(\vec{x_i})| \\ 0 & \textbf{otherwise} \end{cases} \tag{7.1}$$

where

$$\begin{aligned}
\vec{x_i} &= i \text{ th sample location.} \\
f_v &= \text{surface threshold constant.} \\
\alpha_v &= \text{ opacity of voxels having density of } f_v. \\
f(\vec{x_i}) &= \text{density at sample } \vec{x_i}. \\
\bigtriangledown f(\vec{x_i}) &= \text{local gradient vector.} \\
r &= \text{voxel thickness of the transition region.} \\
\alpha(\vec{x_i}) &= \text{opacity at sample } \vec{x_i}.
\end{aligned}$$

What the above equation does is to assign the opacity α_v when the density value is the selected threshold, f_v. It is also desirable to have opacities close to α_v for densities close to f_v. A constant thickness over the transition region also makes a more pleasing image. This is achieved by letting the opacity fall off at a rate inversely proportional to the local gradient vector. When we need to visualize multiple surfaces, each having a different surface constant f_v, each surface can be classified separately using Equation 7.1.

A unit normal is computed as in the Marching Cubes method. A lighting model is then applied at this sample location to compute a color. A running sum of the accumulated opacity is maintained and used to weight the color of each sample location. Processing terminates when the accumulated opacity reaches unity or there are no more voxels left to process. The sum of all the weighted sample colors is the final color for the ray.

Let α_i, $(0 < i < k)$ represent the opacity of the i th sample. Refer to Fig. 7.2. Since x_0 is the first sample, there is no material of the volume that is blocking it, and contribution from this sample is $\alpha(x_0) * I(x_0)$, where $I(x_0)$ is obtained by a local lighting model like that described by Equation 2.8.

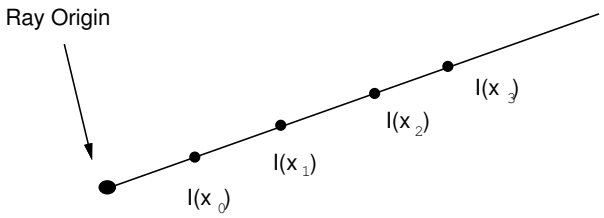


Figure 7.2: Computing Intensity of a Ray.

For the next sample, the amount of material of the volume blocked is $\alpha(x_1)$. The transparency of the volume at sample location x_1 is $(1 - \alpha(x_0))$, and the intensity contribution from sample location 1 is therefore $I(x_1) * \alpha(x_1) * (1 - \alpha(x_0))$. For the third sample, the intensity is $I(x_2)*\alpha(x_2)*(1-\alpha(x_0))(1-\alpha(x_1))$ and so on.

So the total intensity is given by

$$\begin{aligned} I &= I(x_0)\alpha(x_0) + \\ &\quad I(x_1)\alpha(x_1)(1-\alpha(x_0)) + \\ &\quad I(x_2)\alpha(x_2)(1-\alpha(x_0))(1-\alpha(x_1)) + \ldots\ldots \\ &= \sum_{i=0}^{k-1} I(\vec{x_i})\alpha(\vec{x_i}) \prod_{j=0}^{i-1}(1-\alpha(\vec{x_j})) \end{aligned} \tag{7.2}$$

where

$$\begin{aligned} \vec{x_i} &= i \text{ th sample.} \\ I(\vec{x_i}) &= \text{Intensity at sample } \vec{x_i}. \\ \alpha(\vec{x_i}) &= \text{opacity at sample } \vec{x_i}. \\ k &= \text{total number of samples along the ray.} \end{aligned}$$

7.3 Building the k-d Tree

As indicated in the introduction, our goal is to create a data structure which represents interesting volume data in a way that supports fast ray tracing. We do this in two steps.

In the first step, we design a culling function that identifies voxels that can be eliminated from any consideration since they do not contribute to the current view. This step depends on the opacity function used. We demonstrate a culling function to be used when Equation 7.1 computes the opacity. The output of this step is a list of voxels representing the characteristic that will be visualized.

Next we build the *k-d* tree similar to the procedure that we followed for surface models. The partitioning is highly flexible in adapting the partitioning planes to the distribution of the voxels in the hierarchy. For early detection of rays that do not intersect any interesting voxels, bounding volumes are stored at nodes of the hierarchy to make the data structure even more compact.

The *k-d* tree will be used to efficiently access the voxels of interest during ray tracing. The remainder of this section is devoted to a detailed description of the building of the data structure. The next section describes its use in ray tracing.

7.3.1 Identifying 'Relevant' Voxels

The inequalities at the right of Equation 7.1 are the key to determining a culling function for identifying voxels of zero opacity. The culling function

is given by

$$\{(f_{max}(vox_{i,j,k}) \quad + \quad r|\bigtriangledown f_{max}(vox_{i,j,k})|) < f_v\}$$

OR

$$\{(f_{min}(vox_{i,j,k}) \quad - \quad r|\bigtriangledown f_{max}(vox_{i,j,k})|) > f_v\}$$

where

$$\begin{aligned} f_{min}(vox_{i,j,k}) &= \text{min. voxel density at } (i,j,k). \\ f_{max}(vox_{i,j,k}) &= \text{max. voxel density at } (i,j,k). \\ \bigtriangledown f_{max}(vox_{i,j,k}) &= \text{max. voxel gradient at } (i,j,k). \end{aligned}$$

If the above function is **TRUE** for a voxel at (i, j, k), it is discarded; otherwise, it is added to the list of relevant voxels.

Since linear interpolations are used for determining densities as well as gradients within each voxel, the range of densities within each voxel as well as the maximum density gradient of each voxel can be determined. This immediately lets us design a culling function which checks to see if any density in a voxel is within the range of density values that could possibly contribute to the final image. That is exactly what the inequalities in the above function test for. A voxel is irrelevant if its range of densities does not contain the surface threshold density and its density and gradients are such that the opacity function lies completely outside its range. The two parts to the function (on either side of the OR operator) consider the density ranges on either side of f_v, the selected threshold.

This process is repeated for each voxel in the grid and a list of 'relevant' voxels is recorded. All other voxels are not of interest until a different feature or characteristic needs to be studied. The output from the preprocessing step is a list of voxels representing the characteristic of interest.

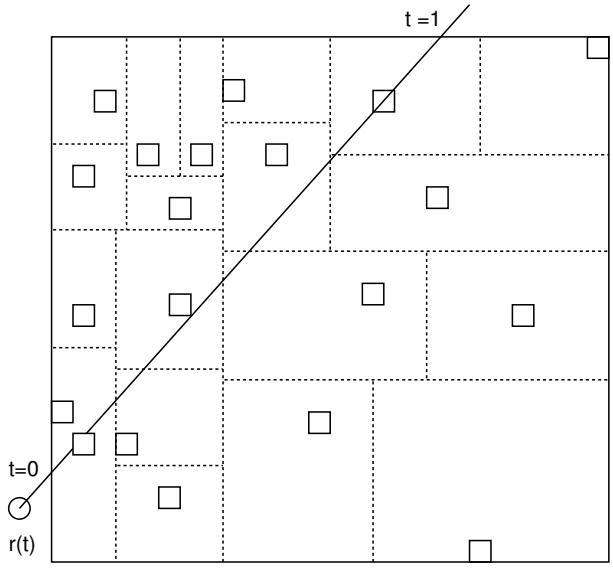


Figure 7.3: A k-d Tree Subdivision.

7.3.2 Building the k-d Tree Hierarchy

Having identified the voxels of interest, the next step is to build a hierarchy to store them. The procedure follows the construction of the *k-d* tree for surface models, described in Section 6.3 and the algorithms in 6.2 and 6.3. As before, two binary searches are performed in each dimension to determine the object and spatial medians. A linear search is performed between the two medians to determine the partitioning plane that minimizes Equation 6.2.

The main difference here is that we are dealing with volume elements instead of surface primitives like polygons or spheres. Also, voxels cannot straddle a partitioning plane since they are aligned with all the partitioning planes. An example is shown in Fig. 7.3 with five levels of partitioning. The dotted lines are partitioning planes and the rectangles represent voxels.

Each time we partition a set of voxels, we need to determine if bounding volumes are required to cull void space. A 2D example is shown in Fig. 7.4.

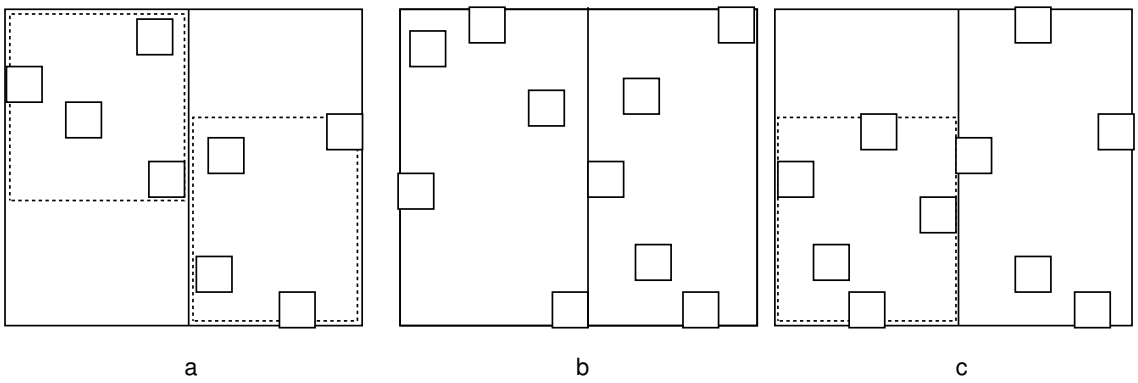


Figure 7.4: Using Bounding Volumes.

In 7.4a, bounding volumes are required on both sides of the plane. In 7.4b, no bounding volumes are required as this results in no reduction in void space, whereas in 7.4c only the left side needs it. The reasoning behind using bounding volumes is that many of the rays will intersect the void areas without intersecting the surface areas, in which case they can be quickly eliminated from further consideration with a simple bounding volume intersection test.

Some important optimizations in this preprocessing step include the following:

When we are dealing with regular grids (common in medical imaging and molecular modeling applications), partitioning planes need be located only on voxel boundaries, which means the entire search can be performed using integer arithmetic.

Since the partitioning planes are axis-aligned and there are only a finite number of locations for each of them, voxels that fall in each of these locations can be summed up. These partial sums can be used to determine the best plane choice. Thus, when we need to determine a plane that best balances the voxels in a region, the partial sums are first computed parallel to the orientation of the plane. These sums can be used to determine the best

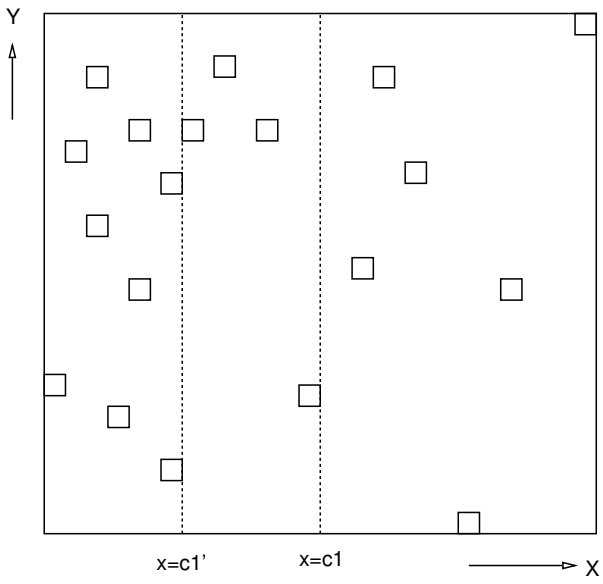


Figure 7.5: Optimizing the Preprocess.

plane as the binary search is conducted. In Fig. 7.5, vertical columns of voxels are added up when we are determining a plane orthogonal to the horizontal axis. In 3-D, if we are searching along the X axis, voxel counts in YZ plane will be summed up at each possible location of the partitioning plane.

A similar strategy can be followed for determining bounding volumes. It is not necessary to consider the entire set of voxels for each location of the partitioning plane. We can precompute bounding volumes of voxels for each location of the partitioning plane. For instance, when the search is along the X axis, bounding volumes of voxels in the YZ plane (these can be thought of as 2D bounding volumes each 1 voxel thick) for each location of the partitioning plane are precomputed. During the plane search, these bounding volumes can be used to determine the actual 3D bounding volume on either side of the plane.

During the binary search, when we move the plane towards a location

that tends to give a better balance of voxels, the side we are moving away from need not have its voxels examined any further. Thus the total number of voxels in this region just needs to be remembered and taken into consideration when the figure of merit is computed. In Fig. 7.5 the plane $x = c_1$ results in a partition with more voxels to its left ($l_{cnt} > r_{cnt}$). The next plane choice is at $x = c'_1$, midway between c_1 and the left bound. To determine the number of voxels on either side of $x = c'_1$, its enough to examine the voxels between the left bound and c_1. The number of voxels to the right of c_1 is r_{cnt} and just needs to be remembered. A similar argument holds if $l_{cnt} < r_{cnt}$.

7.4 Casting a Ray

The ray traversal algorithm follows the description in Section 6.2.5 and the algorithm outlined in Table 6.2.5.1 and will not be repeated here. One difference when rendering volume models of regular 3-D grids is that voxels do not straddle partitioning planes and hence, the sampling process is always ordered along the path of the ray, unlike surface models (refer to Fig. 6.5) where a primitive intersection point could turn out to be outside the region being examined. Each time a leaf node is encountered during the traversal process, a sample is picked midway between the intersections of the ray with the voxel. The contribution of this sample is integrated using Equation 7.2.

7.5 Implementation and Results

We have implemented our algorithm in C on an Ardent Titan 1500 running Ardent UNIX[1] 2.1.1 and tested the implementation on three different

[1]UNIX is a trademark of AT&T Bell Laboratories

Scene	Total Voxels	Relevant Voxels	% Reduction	Memory (Mbytes)
Heart	1245184	61681	95.0	4.5
Heart(sliced)	1245184	38198	96.9	2.5
SOD at 80	1091444	63533	94.2	3.0
SOD at 100	1091444	27427	97.5	1.3
HIPIP	262144	24628	90.6	1.2

Table 7.1: Voxel Statistics.

Scene	Timings (minutes)					
	median-cut			k-d		
	Preprocess		Render	Preprocess		Render
	Cull	Build		Cull	Build	
Heart	2.73	0.28	6.26	2.73	0.74	5.78
Heart(sliced)	2.38	0.17	5.93	2.38	0.45	5.56
SOD at 80	2.48	0.29	11.60	2.47	0.78	11.43
SOD at 100	2.33	0.13	8.09	2.33	0.33	7.48
HIPIP	0.61	0.10	6.75	0.62	0.28	6.51

Table 7.2: Timing Statistics.

test cases. The first case is an MRI data-set of a cadaver heart that has been autopsied; during autopsy, cuts were made into the ventricles. The heart was in a bucket of preservative and was imaged from three orthogonal directions. The data consists of 76 slices, each of resolution 128x128 Color plates 9.11a and 9.11b show two different views of the heart. Next the heart is sliced vertically (by a plane) and the voxels in front of the slicing plane are removed so as to get a better view of the heart chambers. Plates 9.12a and 9.12b are the result.

The next example is an electron density map of the active site of superoxide dismutase (SOD) enzyme as determined by x-ray crystallography at 1.8 angstrom resolution. The data-set consists of 116 slices, each of size 97x97. Plate 10.13 shows one frame of an animation sequence when centered around a threshold of 80. At this level, teardrop shapes and clumps of atomic density are seen. At a level of 100, we start seeing individual atoms, as illustrated in

plate 9.14.

The last example is a quantum mechanical calculation of one electron orbital of a four-iron, eight-sulphur cluster found in many natural proteins. This particular data-set is a high potential iron protein (HIPIP). The data represents the scalar field of the wave-function at each point. The resolution of the data is 64x64x64. Scientists are interested in seeing 'nodal' surfaces, where the data value crosses zero. Plate 9.15 attempts to demonstrates this.

Table 7.1 illustrates the reduction in voxel data after the preprocess. In all cases, less than 10% of the total voxels are relevant to the final image. This demonstrates the importance of culling irrelevant voxels and working with only voxels of interest. In the heart images, slicing causes further culling of the data. All timings are for a resolution of 640x480, with one ray cast per pixel. In this implementation, we have not taken advantage of either vectorization or parallelization to optimize performance.

Table 7.2 shows the run times for all the test cases. Two sets of timings are shown; the first using the median-cut partitioning strategy and second, timings using the k-d tree search structure. The preprocess times using the k-d tree are a little higher because of the extra work involved in the hierarchy construction. However the rendering times are faster using the k-d tree (for all the test cases). These savings will accumulate in a long animation sequence since the search structure needs to be constructed only once.

In order to compare our method with the traditional algorithm which directly samples the three dimensional grid, we have an implementation that does not perform any culling or use any form of space partitioning. Results of rendering the same datasets are shown in Table 7.3. In all these cases, there is at least a factor of 20 or more in performance advantage with the use of the

Scene	Time (min)	
	Preprocess	Render
Heart	0.90	289.64
Heart(sliced)	0.88	226.68
SOD at 80	0.45	228.57
SOD at 100	0.45	278.75
HIPIP	0.11	98.51

Table 7.3: Results Using the Traditional Algorithm.

k-d tree partitioning.

We are presently adapting this technique to be useful for users located remote from the supercomputer center. Our strategy is to preprocess the data, whereby the voxels representing a given characteristic is identified. This set of voxels is then transmitted over the network to the researcher's workstation for rendering. In filtering the voxels of interest from the original three-dimensional grid, a large reduction in the number of voxels transmitted over the network will result in a natural compression of the original data-set. Since building the *k-d* tree search structure usually takes a fraction of the rendering time, it can be performed remote from the supercomputer center, thus making it unnecessary to transmit the search structure over the network.

7.6 Conclusions

We have presented an algorithm which uses space partitioning techniques to support efficient volume rendering. A cull function prunes voxels from the original data that are irrelevant to the characteristic being studied. A search structure based on *k-d* trees is used as a data structure for storing the relevant voxels so that they may be accessed efficiently during rendering. This data structure has important advantages for rendering volumetric data:

1. It provides a compact representation of voxels. The partitioning planes in the hierarchy help in determining a traversal order that identifies only voxels close to the path of each ray. The bounding volumes used in the internal nodes of the hierarchy help in identifying rays that do not intersect any relevant voxel by simple bounding volume intersection tests.

2. Since the tree can be traversed along the path of a ray, the search can be terminated once the accumulated opacity reaches unity.

3. The search structure is direction independent. With the help of the partitioning planes, a traversal order can be determined for any ray with arbitrary origin and direction. Changes in viewing or lighting parameters require no change in the search structure.

4. The choice and location of the partitioning planes is flexible. The plane that best balances the voxel counts on either side of the partition is chosen. The plane can be aligned with any of the three dimensions.

5. Since partitioning planes need to be located only on voxel boundaries, the entire plane search can be performed using integer arithmetic.

Surface modeling approaches to visualizing volumetric data work best when few surfaces are involved. Existing ray-tracing approaches exploit the regular nature of three-dimensional grids through the use of incremental techniques for volume traversal. These techniques are most efficient when the voxels of interest are distributed sufficiently densely through the volume. In the data that we have worked with, this has not been the case. Visualizing multiple features increases the number of voxels participating in the view, but there is a point beyond which visualizing multiple characteristics (thereby increasing

the number of relevant voxels) only makes it increasingly difficult to interpret the data. Our approach is targeted at producing images of intermediate complexity, i.e. those which may contain more than one type of surface but not so much relevant data that the majority of the volume is involved. The technique should thus be complementary to existing incremental ray-tracing approaches which work best on very dense data and surface modeling techniques which work best on data with few surfaces of interest.

Chapter 8

CONCLUSIONS

Although ray tracing is one of the most popular rendering techniques for realistic image generation, its major difficulty has been its demand for tremendous computational resources. This has prompted a great deal of research into developing novel search structures for accelerating ray tracing.

Most of the search structures in use today take advantage of the characteristics of the input scene in different ways. They also have constraints in their construction, leading to inefficiencies in the data structure. The performance of existing data structures is evaluated purely by timing benchmarks. If a search structure performs poorly on a given input scene, then there is no mechanism for detecting this prior to rendering. A key advantage to having some idea of the performance before rendering is that a different search structure may be substituted for better performance.

In this dissertation, we have presented new results for significantly increasing the adaptivity of search structures to scene characteristics leading to improved performance of ray tracing. We have shown the importance of the knowledge of a search structure's performance prior to rendering. A cost model has been built and applied to a variety of search structures (both hierarchical and non-hierarchical) based on space partitioning and bounding volumes. The model is built using statistical properties of the search structures. What is more

important is that the model evaluation can be built into the preprocessing step, thus permitting a technique's performance to be evaluated before rendering.

The model has been demonstrated to be a very effective means for predicting automatic termination criteria for existing search structures such as the BSP tree, octree, ABV hierarchy and the uniform subdivision techniques. This has improved the performance of all these search structures, at the same time making it unnecessary to set termination conditions in advance without any knowledge of the input scene characteristics. Experimental results have been presented to validate the effectiveness of the model in solving this problem.

Secondly, the model has also been found to be reasonably accurate in choosing a search structure for ray tracing a particular input scene. While the assumptions in the cost model cause it to make wrong decisions, it has been shown that it is useful in performance improvement by virtue of the fact that the model never picks the worst method on all the test cases.

An extensive experimental study has been made of some important properties of search structures based on space partitioning as well as those based on bounding volumes. Characteristics such as location and orientation of partitioning planes, effects of bounding volumes in search structures, presence of void spaces at the nodes of search structures, size of the hierarchy (for example, its average height) and hierarchy traversal methods have been examined in detail.

A combination of the advantageous characteristics of all these search structures has led to the development of the k-d tree, a new search structure for accelerating ray tracing. Its advantages stem mainly from its greater flexibility in its construction. The partitioning planes are axis-aligned, but they can be located anywhere within the region extent and oriented along any of the

principal axes. The void spaces that are created by this partitioning are pruned by using simple bounding volumes at nodes of the hierarchy where its reduction results in a significant performance advantage.

As an example of its adaptivity to a variety of scene models, the k-d tree structure has been used successfully to render volumetric models from scientific applications. Models from such diverse applications as medical imaging, molecular modeling and x-ray crystallography have been rendered with this search structure. The benefits of space subdivision techniques in rendering volumetric data are especially important when the features of interest occupy a small fraction of the original data and can be extracted from it efficiently during the preprocess. In the scientific applications mentioned, this has been found to be the case. Thus, it is highly beneficial to understand the characteristics of the data set prior to rendering. For instance, if it is determined that most of the original volume model participates in the rendering, then it can be quickly determined that the benefits of space subdivision techniques are less advantageous and hence, its use not recommended. In this case, the traditional ray casting technique could be substituted for better performance.

8.1 Future Work

An important parameter involved in the evaluation of the cost model is the expected number of regions visited by each ray. This quantity is computed with the knowledge of the region hitting probability of any node in the search structure, given that the ray intersects the scene extent. Under the assumption that rays are uniformly distributed, this probability was shown to be directly related to the area of the convex hull of the objects stored at node of the hierarchy.

The cost model that we have developed while being adequate for predicting termination criteria, is not accurate enough for predicting the computational requirements necessary for rendering a particular scene. A more sophisticated model is required to determine accurate cost values to match experimental results. For this, distributions of the ray directions and those of the primitives in the scene have to be taken into account. Void spaces in collections of objects surrounded by bounding volumes have to be taken into account when computing the hitting probabilities. The model assumes that if the bounding volume is penetrated by an incoming ray, an object intersection is found within the cluster of primitives contained by the volume.

In our work, study of scene characteristics has been through the investigation of parameters that directly affect the construction of object clusters by different search structures. Another approach that might be worth looking at is to directly start at the object primitive level and examine various object distributions. How search structures will handle or partition such object distributions could give valuable information as to their characteristics and as a result, performance.

Direct volume rendering using a ray tracing approach is becoming a common technique for sampling 3D scalar fields. While we have shown a solution using space subdivision techniques, much remains to be done. We mentioned earlier that space subdivision structures like the k-d tree provide an efficient representation of volumetric data. Before they can be used to render the volume models, the voxels in the volume data need to be classified as interesting or not. This is not an easy problem and is being actively investigated. The problem here is no one technique can prove to be the best. This is because volume datasets come from a variety of applications and thus could represent

characteristics that are very different from each other.

Regular 3D grids are the only kind of volumetric data that has been investigated so far. However, irregular grids are widely used in applications such as computational fluid dynamics. We are currently extending the use of the *k-d* tree to render data from such applications as computational fluid dynamics where the data is usually not in a regular grid. This looks very promising since the the high degree of adaptivity of the *k-d* tree makes it a good candidate for visualizing data stored in irregular grids.

Chapter 9

COLOR PLATES

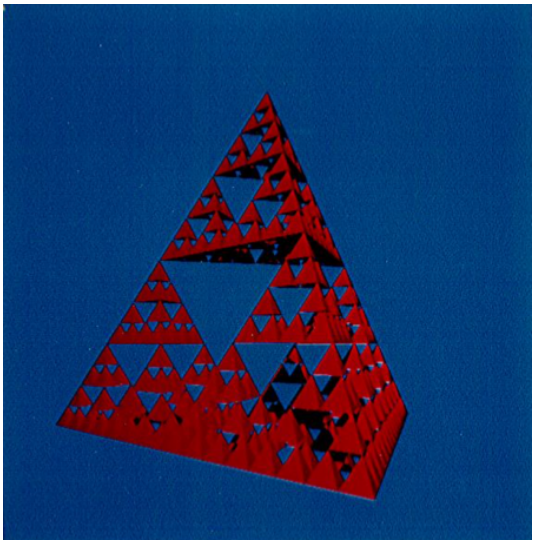


Plate 9.1. Tetrahedra

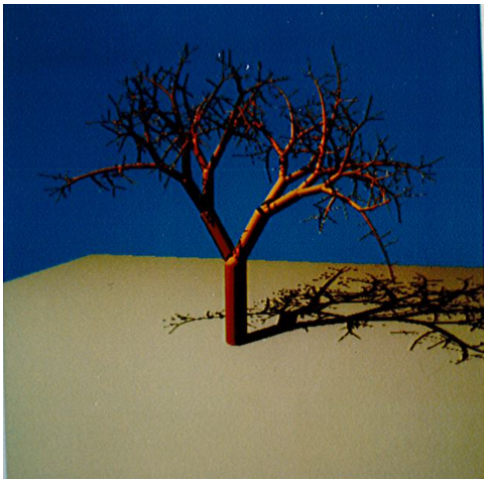


Plate 9.2. Caltech Tree (Branches)

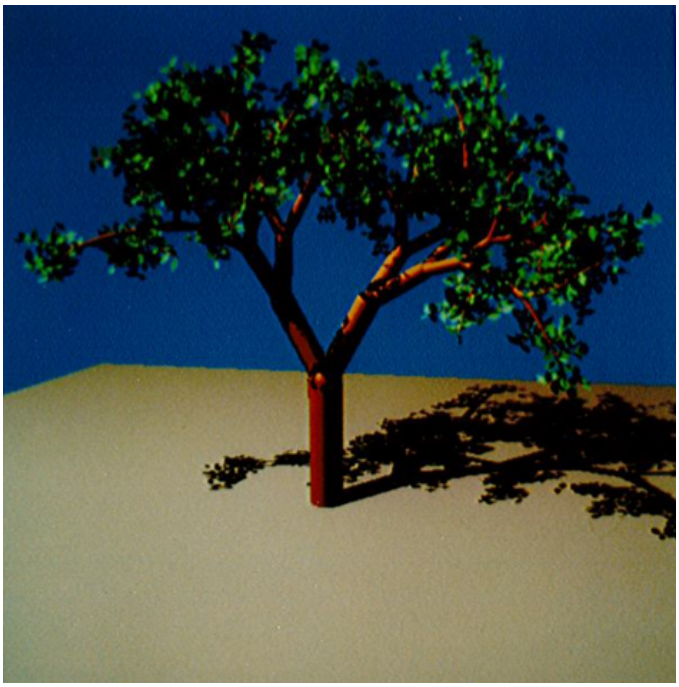


Plate 9.3. Caltech Tree

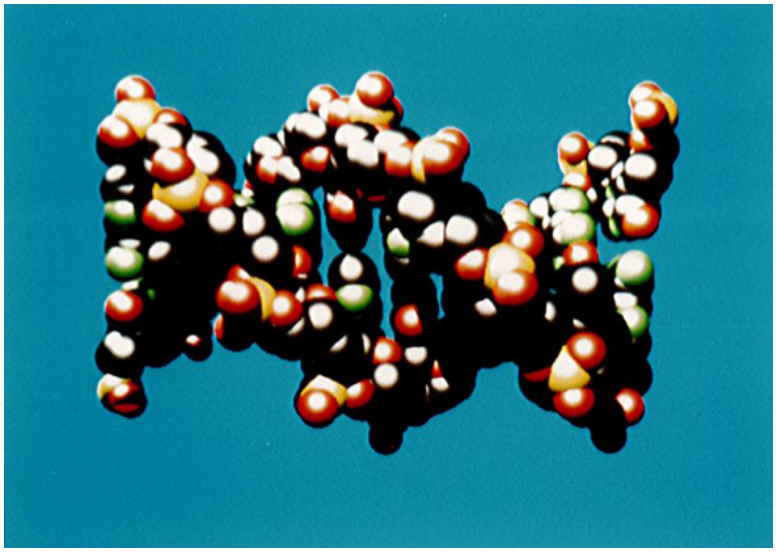


Plate 9.4. DNA

Plate 9.5. Teapot

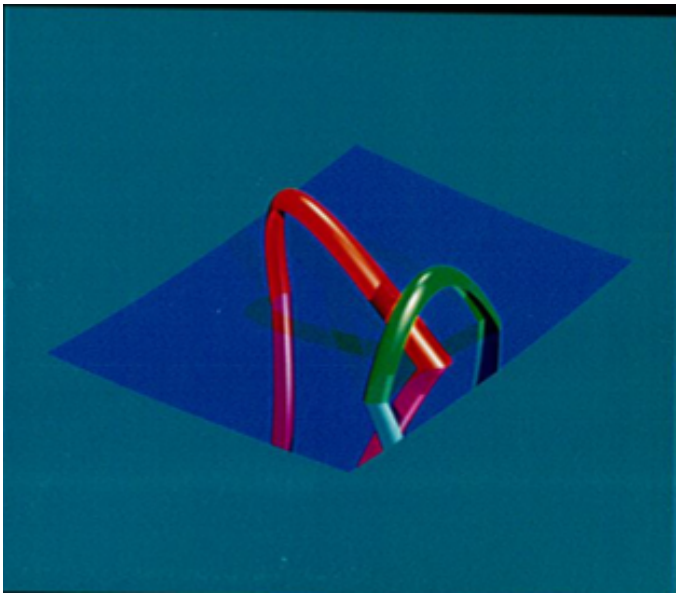


Plate 9.6. Arches

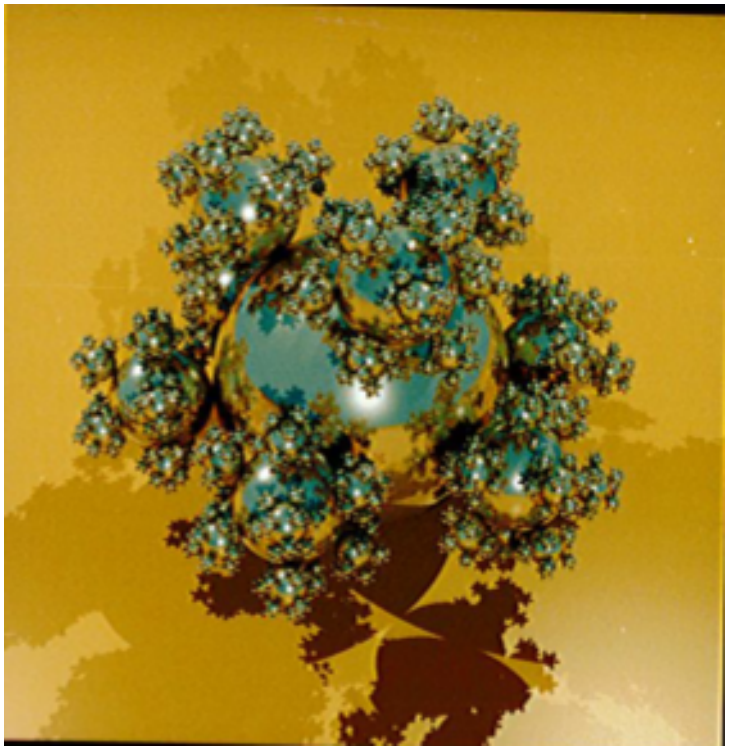


Plate 9.7. Spd Balls

Plate 9.8. Spd Tree

Plate 9.9. Spd Mountain

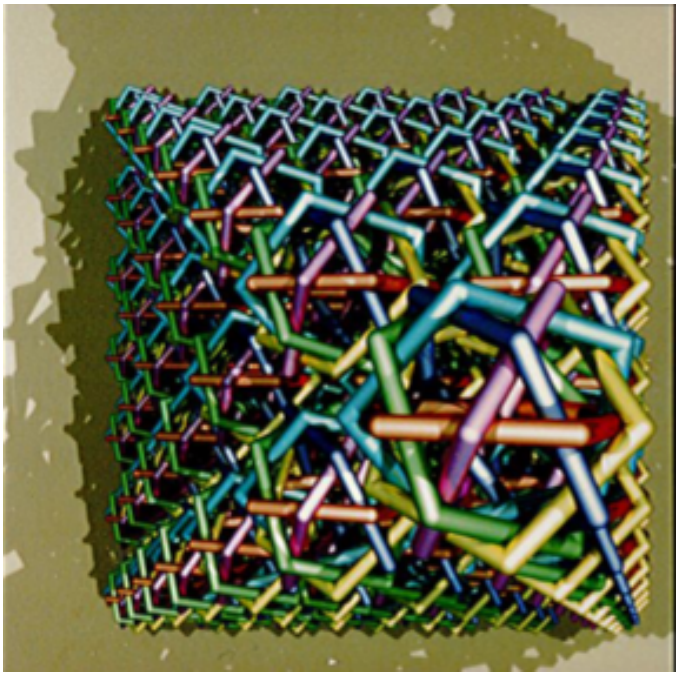


Plate 9.10. Spd Rings

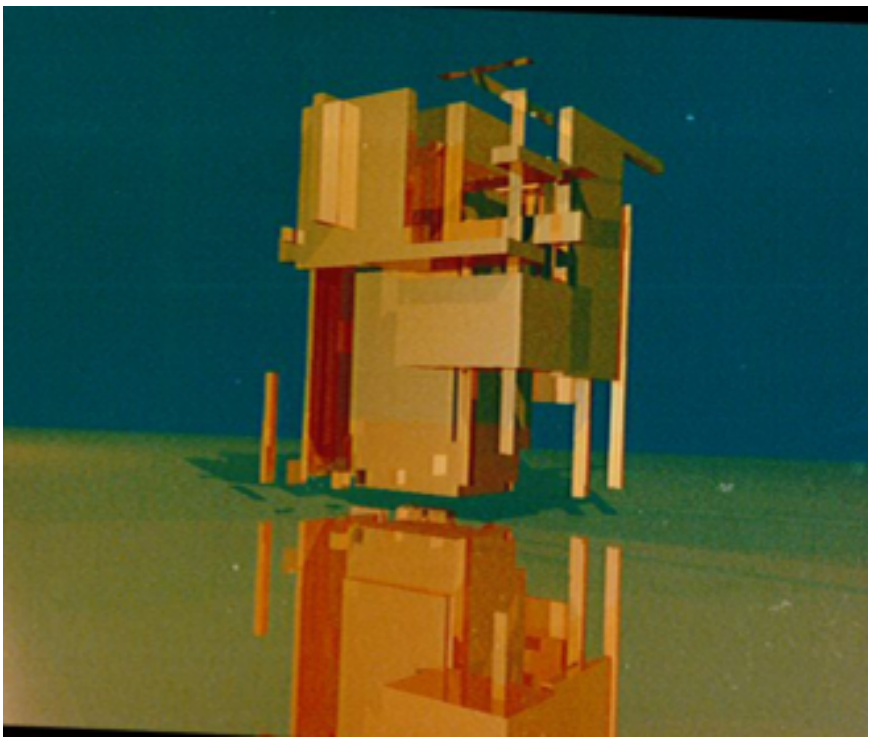


Plate 9.11. Geo 58

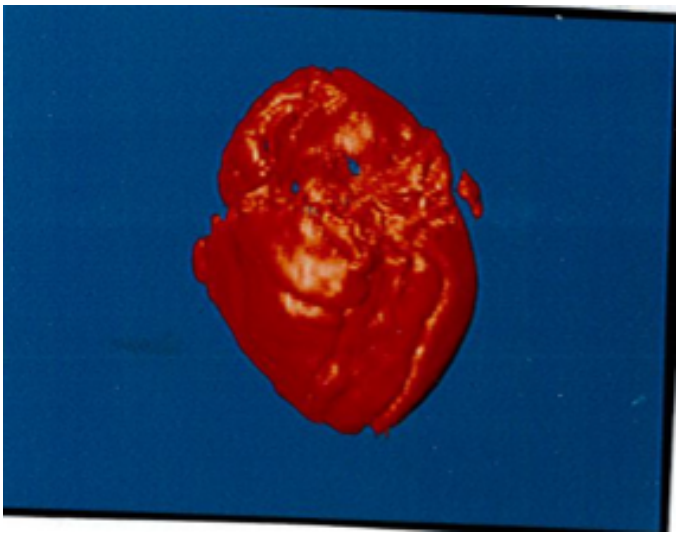


Plate 9.12. Heart (View 1)

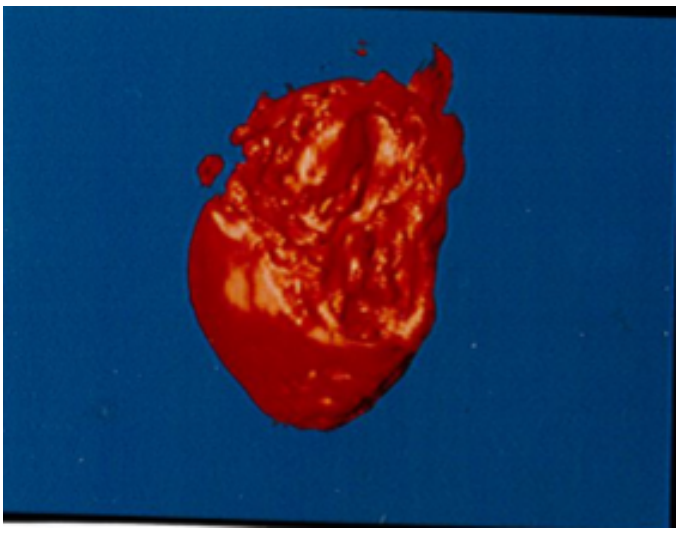


Plate 9.13. Heart (View 2)

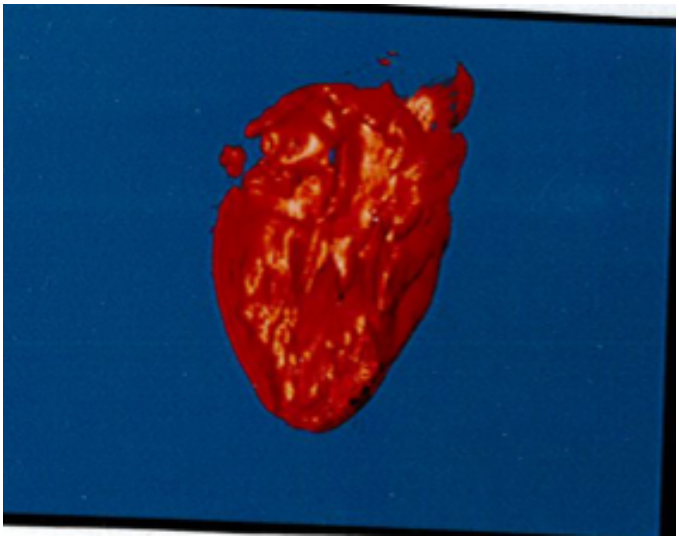


Plate 9.14. Heart (Sliced View 1)

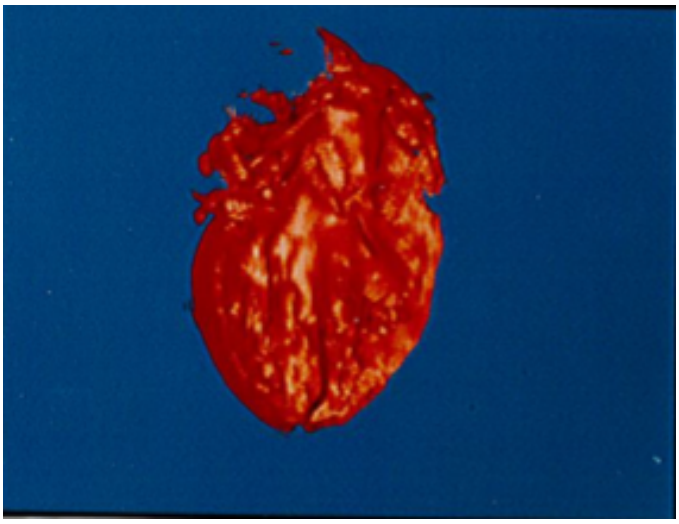


Plate 9.15. Heart (Sliced View 2)

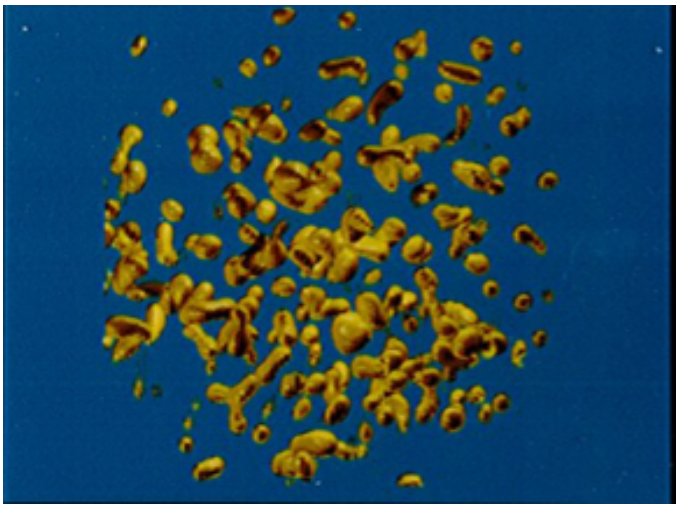


Plate 9.16. SOD (Isovalue = 80)

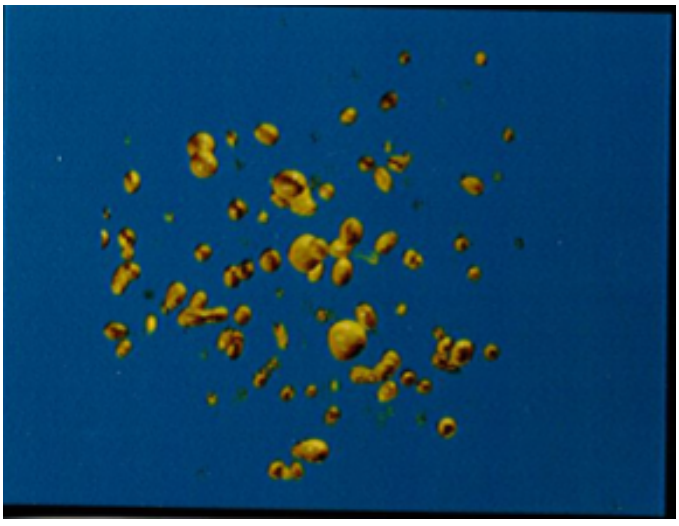


Plate 9.17. SOD (Isovalue = 100)

Plate 9.18. HIPIP (Nodal Surfaces)

BIBLIOGRAPHY

[1] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics*, 21(4):269–278, July 1987.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), September 1975.

[3] Jon Louis Bentley. Data structures for range searching. *Computing Surveys*, 11(4), December 1979.

[4] C. Buckalew and D.S. Fussell. Illumination networks: Fast realistic rendering with general reflectance functions. *Computer Graphics*, 23(3):89–98, August 1989.

[5] Norman Chin and Steven Feiner. Near real-time shadow generation using bsp trees. *Computer Graphics*, 23(3):99–106, July 1989.

[6] J.G. Cleary and G. Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Visual Computer*, 4(2):65–83, July 1988.

[7] M.F. Cohen, S.E. Chen, J.R. Wallace, and D.P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics*, 22(4):75–84, August 1988.

[8] M.F. Cohen and D. Greenberg. The hemicube: a radiosity solution for complex environments. *Computer Graphics*, 19(3):31–40, August 1985.

[9] R.L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[10] R.L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):137–145, July 1984.

[11] R.L. Cook and K.E. Torrance. A reflection model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24, January 1982.

[12] R.A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4), August 1988.

[13] H. Fuchs, G.D. Abram, and E.D. Grant. Near real-time shaded display of rigid objects. *Computer Graphics*, 17(3):65–72, July 1983.

[14] H. Fuchs, Z.M. Kedem, and B.F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics*, 14(3):124–133, July 1980.

[15] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, April 1986.

[16] Andrew Glassner. Introduction to ray tracing. *ACM SIGGRAPH Course Notes 7*, August 1988.

[17] A.S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[18] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, pages 14–20, May 1987.

[19] C.M. Goral, K.E. Torrance, D.P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics*, 18(3):213–222, July 1984.

[20] Eric A. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, pages 3–5, November 1987.

[21] Eric A. Haines and Donald P. Greenberg. The light buffer: A shadow testing accelerator. *IEEE Computer Graphics and Applications*, pages 6–16, September 1986.

[22] Eugene Hecht and Alfred Zajac. *Optics*. Addison Wesley Publishing Company, Reading, Massachusets, 1985.

[23] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. *Computer Graphics*, 18(3):119–127, July 1984.

[24] J.T. Kajiya. The rendering equation. *Computer Graphics*, 20(4):143–150, August 1986.

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

[26] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, August 1986.

[27] Jeffrey M. Lane, Loren C. Carpenter, Turner Whitted, and James F. Blinn. Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM*, 23(1), January 1980.

[28] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3), May 1988.

[29] M. Levoy. Design for a real-time high-quality volume rendering workstation. In *Chapel Hill Workshop on Volume Visualization*, pages 85–92. Computer Science Dept. of the University of North Carolina, 1989.

[30] M. Levoy. A hybrid ray tracer for rendering polygon and volume data. *IEEE Computer Graphics and Applications*, 10(2), March 1990.

[31] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface reconstruction algorithm. *Computer Graphics*, 21(4), July 1987.

[32] J. David Macdonald and Kellog S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(3), June 1990.

[33] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, June 1982.

[34] B.F. Naylor. *A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes*. PhD thesis, The University of Texas at Dallas, May 1981.

[35] W.M. Newman and R.F. Sproull. *Principles of Interactive Computer Graphics*. McGraw-Hill, 1979.

[36] B.T. Phong. Illumination for computer generated images. *Communications of the ACM*, 18(2):311–375, 1975.

[37] Gordon W. Romney. *Computer Assisted Assembly and Rendering of Solids*. PhD thesis, Dept. of Electrical Engineering, University of Utah, Salt Lake City, Utah, 1969.

[38] S.D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144, February 1982.

[39] S.M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, 1980.

[40] Robert A. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, U. S. Air Force Human Resources Laboratory, September 1969.

[41] D.S. Shlusselberg and W.K. Smith. Three dimensional display of medical image volumes. *NCGA 86 Conf. Proc. NCGA, Fairfax, Virginia*, 1986.

[42] R. A. Shumacker. Study for applying computer generated images to visual simulation. Technical Report AEHRL-TR-69-14, (AD700375), US Air Force Human Resources Lab, March 1969.

[43] J.M. Snyder and A.H. Barr. Ray tracing complex models containing surface tesselations. *Computer Graphics*, 21(4):119–128, July 1987.

[44] L. Richard Speer, Tony D. DeRose, and Brian A. Barsky. A theoretical and empirical analysis of coherent ray tracing. *Graphics Interface*, pages 11–25, May 1985.

[45] L. Stone. *Theory of Optimal Search*, pages 27–28. Academic Press, New York, 1975.

[46] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterisation of ten hidden-surface algorithms. *Computing Surveys*, 6(1), March 1974.

[47] W.C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics*, 21(4):153–162, July 1987.

[48] Kelvin Thompson and Donald S. Fussell. Amalgamated ray tracing, inc. To be submitted.

[49] C. Upson and M. Keeler. Vbuffer:visible volume rendering. *Computer Graphics*, 22(4), August 1988.

[50] J.R. Wallace, M.F. Cohen, and D.P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity models. *Computer Graphics*, 21(4):311–320, July 1987.

[51] J.R. Wallace, K.A. Elmquist, and E.A. Haines. A ray tracing algorithm for progressive radiosity. *Computer Graphics*, 23(3):315–324, August 1989.

[52] G.J. Ward, F.M. Rubinstein, and R.D. Clear. A ray tracing solution for diffuse interreflection. *Computer Graphics*, 22(4):85–92, August 1988.

[53] John E. Warnock. *A Hidden Surface Algorithm for Halftone Picture Representation*. PhD thesis, Department of Computer Science, University of Utah, Salt Lake City, Utah, 1969.

[54] H. Weghorst, G. Hooper, and D.P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[55] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

VITA

Kalpathi Raman Subramanian was born in Kuala Lumpur, Malaysia on May 21, 1960, the son of Mrs. Lakshmi Raman and Mr. Kalpathi S. Raman. He moved to Madras, India in 1967 and completed his high school work at Padma Seshadri Bala Bhavan Matriculation School, Madras in June, 1977. He completed his Pre-University Course at Loyola College, Madras in June, 1978. He received the degree of Bachelor of Engineering (Honors) in Electronics and Communications from College of Engineering, Guindy, Madras, India in June, 1983. In September, 1983 he entered the Graduate School of The University of Texas at Austin, Austin, Texas 78712. In December 1987 he received the degree of Master of Science in Computer Sciences from The University of Texas at Austin.

Permanent address: 100 Kamdar Nagar
North Usman Road
Nungambakkam,
Madras - 600034
India

This dissertation was typeset[1] with LaTeX by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin dissertation format was written by Khe-Sing The.