DEVELOPMENT OF A FAST RAY-TRACING ALGORITHM

MING TAN

ISBN 0-612-06149-3

Development of A Fast Ray-Tracing Algorithm

By

Ming Tan B.Sc.

A thesis submitted to the School of Graduate Studies

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

April, 1995

St. John's Newfoundland Canada

Abstract

Ray-tracing is one of the best rendering techniques for the creation of high quality and realistic computer synthesized 2D images of the 3D world. It has been successfully applied in many graphics packages. However, while the technique renders realistic images by simulating the behavior of real light rays, a ray tracing algorithm also imposes a heavy computational burden. As the most time-consuming part of a ray tracer is to check the intersections between imaginary rays and object surfaces, an efficient algorithm is crucial. Thus, many efficient acceleration algorithms have been invented. Bounding volume techniques are the most popular among the many acceleration techniques for ray/object intersection testing. Although a few bounding volumes have been proposed, spheres are the simplest. In this thesis, two ray/sphere intersection algorithms, i.e. the traditional algebraic solution and the efficient geometric solution, are first discussed. Then, a new ray tracing algorithm is introduced that features a fast ray/sphere interaction. By homogeneously transforming the definition world to a ray, the new algorithm simplifies the expensive 3-D surface intersection problem into a 2-D enclosure check. The speed-up of image rendering is derived theoretically in the thesis from complexity analysis and is also demonstrated in an implementation with experimental results.

Acknowledgments

I would like first to express my thanks to my two supervisors Prof. Xiaobu Yuan and Prof. Paul Gillard for their interest, financial support, guidance and patience during my studies at Memorial University of Newfoundland. Without their help and supervision, it would be impossible to give the thesis its current quality.

I would also like to thank other professors at Department of Computer Science who taught or helped me: Prof. Caoan Wang, Prof. Jian Tang, Prof. Siwei Lu and Prof. John Shieh.

Special thanks are due to staff members who have helped me in one way or another.

Finally I thank those graduate students who provided encouragement and assistance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One goal of computer graphics is to create 'realistic' images. For this purpose, people simulate the real world's optical physics in the virtual world of the computer. Many rendering techniques have been developed, such as hidden surface removal, shadow computation, reflection of light, transparency, motion blur, and global specular interaction. However, most of the rendering algorithms work only in special cases. One often finds a picture with shadows, but no transparency, or another picture with reflection, but no motion blur.

Ray-tracing was introduced as geometric optics in Rene Descartes' treatise, published in 1637 and was used to explain some physical phenomena at that time. Some people suggested introducing this technique for rendering images by computer in the 1960's. It was not feasible then because ray-tracing algorithms take a great deal of time

for calculation. As computers became more and more powerful, graphics researchers began to think that this physical simulation of light would be a good way to create high quality synthetic images of the real world.

In a ray tracing algorithm, a ray is first projected through one pixel in an image backwards into the scene along the viewing direction. At the intersection point of the ray with the nearest surface, the color intensity is computed under the illumination of both the local light sources and two contributions from the reflected and transmitted light intensities. The secondary light contributions are in turn accumulated recursively along the reflected and refracted rays as with the original ray until they are 'aged' off[27].

As a result, a ray tracing algorithm mainly involves two types of processing — surface intersection and intensity calculation. While color intensity can be fairly easily computed from illumination models[28], a ray tracing algorithm spends most of its time on ray-surface intersection tests[1]. Therefore, an efficient ray-surface intersection testing method will definitely accelerate the ray tracing process and thus speed up the rendering of realistic images.

In this thesis, a new ray tracing algorithm is introduced. It employs a fast ray/sphere interaction test that simplifies the expensive three-dimensional surface interaction problem to a simple two-dimensional enclosure check. This ability is obtained by transform-

[1]It was estimated[21] that, for a scene of moderate complexity, a recursive ray tracing algorithm spends up to 95% of its time checking ray-surface intersections.

ing the coordinate system of the scene to a coordinate system related to the direction of the ray. Technical details are presented in the thesis. To demonstrate the speed improvement of image rendering with this new algorithm, computational complexity is analyzed in terms of numerical operations. Comparison with three typical ray-tracing algorithms is also given with results from real implementations.

Chapter 2

Ray-Tracing Processing

2.1 Physical Theory

A ray traced through a scene enables the computation of the intensity of the pixel associated with the ray. Assuming the big and small spheres in Figure 2.1 are completely opaque, and that the cube is partially transparent, actually three rays can be traced in the direction of light propagation:

1. A ray from the light source that reflects off the small sphere and refract through the cube to the pixel of interest.

2. A ray from the light source that reflects directly from the cube to the pixel of interest.

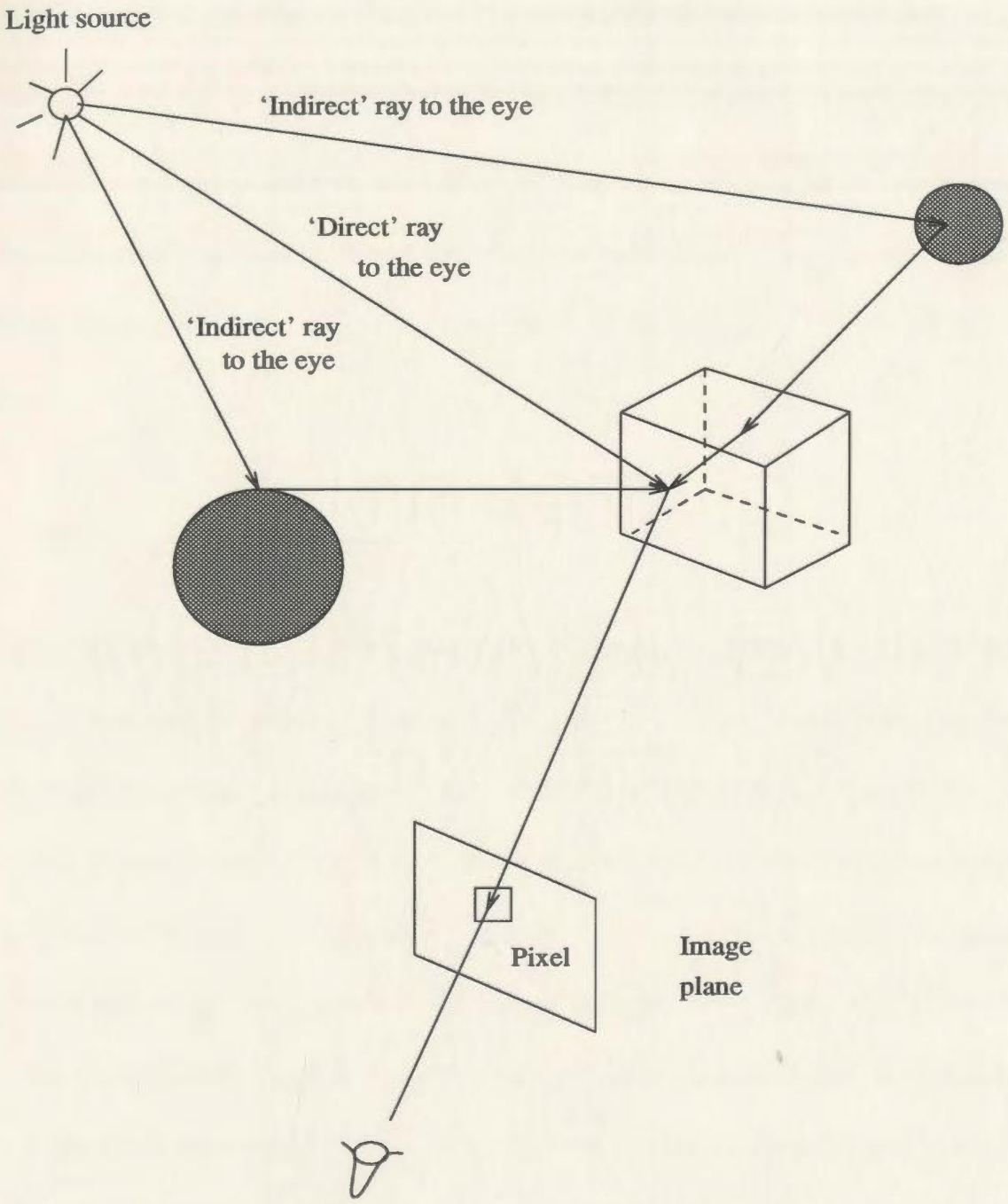


Figure 2.1: Direct and indirect rays that travel from the light source to a pixel

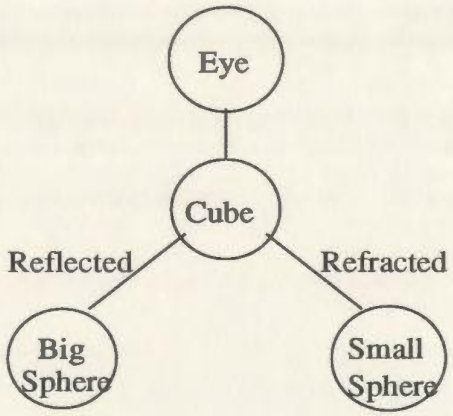
3. A ray from the light source that reflects off the big sphere and then off the cube to the pixel of interest.

The first ray is 'indirect' as the light propagates along four separate paths in its journey from the light source to the pixel. The second ray is 'direct'. Although both rays are technically indirect, a ray that reaches to the eye due to a single reflection is different from ray that is the result of multiple reflections. The third ray is also indirect.

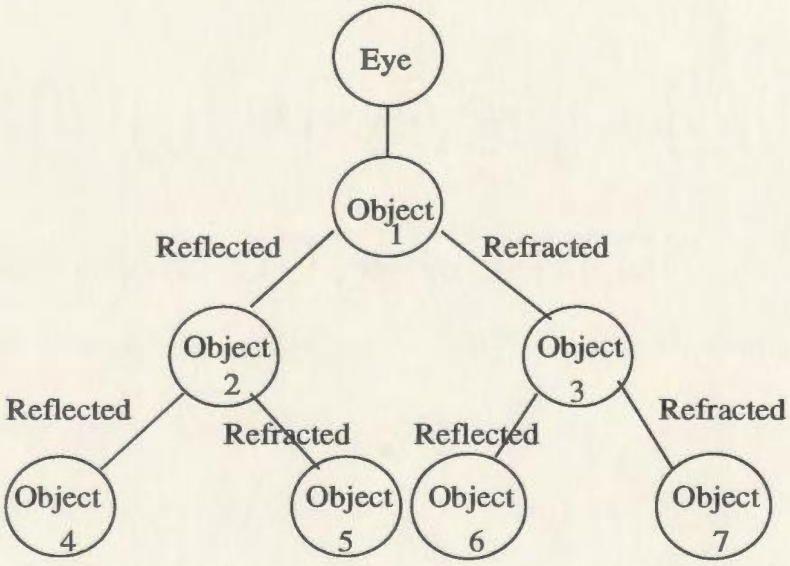
2.2 The Algorithm

Ray tracing is often perceived as a tree creation process. A tree of one pixel in the scene shown in Figure 2.1 is given in Figure 2.2(a). In this figure, each node represents a recursive call of a general ray tracing procedure. Such ray trees are subsets of a general binary tree(Figure 2.2(b)) that will have a refracted and a reflected branch emanating from each node. If a ray intersects an object, it then spawns two rays in general — a reflected ray and a transmitted or refracted ray. Each of these rays produces two rays at the next interface and a recursive trace continues until either a predetermined recursive depth is exceeded, or a ray hits nothing and is allotted a background color. Herein lies the high computational cost of recursive ray tracing.

In any implementation, it usually traces rays *backwards* from the view point through



(a) Tree for a ray shown in Figure 2.1



(b) Ray trees are subsets of a general tree.

Figure 2.2: Ray tracing as a tree.

each pixel into the scene. That is, it traces in the reverse direction of light propagation. Since an infinite number of rays emanate from a light source and it is difficult to trace rays in the direction of light propagation, we only trace those rays which could pass through the image plane among all the rays within the 3D scene because they are the rays contributing to the picture intensities.

In a naive ray tracer, a ray is tested against every object in the scene for intersection[20]. If more than one object intersects the ray then the one that is the nearest to the ray origin is selected.

At each intersection point, the light intensity is calculated as follows:

$$I = I_l + k_r I_r + k_t I_t$$

That is, the light reflected from a point on a surface is the linear combination of three terms. First, a local term I_l counts in the local illumination at the intersection point on the surface directly from light sources. It can be calculated by using any of the direct reflection model descriptions, e.g., the Phong model[22]. Secondly, the global terms, I_r and I_t, are added into the local component. I_r represents light that arrives at a point due to reflection from another object, and I_t light that transmits through the object if it is partially transparent. Here k_r and k_t are reflection and transmission coefficients relevant to the surface material.

It is important to bear in mind that this is a recursive process. A tree is evaluated bottom-up at every intersection point using the same equation to calculate the intensity, and each node's intensity is computed as a function of its children's intensities.

The algorithm involves two stages of processing. The first stage is the ray/object intersection testing. In the second stage, the ray is recursively traced and the intersection point light intensity is calculated. It is also noted that the first part is separate from the second part. This means that the calculations for ray/object intersection testing are just for determining the intersection point location in the scene and is totally processed before the actual intersection point light intensity evaluation.

This is a well-known important advantage of the ray tracing approach. This advantage makes it possible to independently develop totally different ray/object intersection strategies for the ray tracing algorithm because the new intersection algorithm is easy to add into the ray tracer. As mentioned in the introduction, the ray tracer will spend most of its calculation time on intersection tests. Therefore, a better intersection checking strategy can always be used to speed up the ray tracing process.

Chapter 3

Survey of Ray Tracing Acceleration Techniques

3.1 Introduction

The ray-tracing algorithm depends exclusively on a single operation: calculating the point of intersection between a ray in the 3-D space and a geometric object, or *primitive*. The primitive objects could have any shape, e.g. polygonal, spherical, cylindrical, and more complex shapes like parametric surfaces [12, 13] or swept surfaces[15]. Rays have to do the intersection operations with the collection of all possible primitive objects, which actually define an *environment*.

Each ray must be tested with all the primitive objects within the environment

and the intersection point is the point closest to the ray origin. This is commonly referred to as the 'standard' (or 'traditional') ray-tracing algorithm. This has a linear time complexity in the number of objects. This *exhaustive* ray tracing is the most intuitive solution, and it is still employed in the processing of subproblems within more complicated techniques because some complicated environment might prevent the use of the acceleration techniques.

Accounting for the huge time consumed by the ray tracing algorithm, it is found that the cost of the operation for calculating the intersection point typically overshadows other operations in the ray-tracing algorithm such as intensities evaluations and common book-keeping operations. A statistic reported by Whitted[21] is that more than 95% of the time could be spent conducting this operation for scenes of moderate complexity.

Most techniques to accelerate ray tracing fall into one of three categories: reducing the number of objects for a given inclusion test, providing more efficient intersection tests for objects and bounding volumes, and simultaneous multiple-ray projection schemes (e.g., beam tracing[35] and cone tracing[36]). The first two techniques can often be used simultaneously, but may require that "pure" ray tracing be employed — in "pure" ray tracing, intersections are calculated with a single ray.

The following survey examines some of the strategies employed for the first two of the previous acceleration categories, employing "pure" ray tracing.

3.2 Bounding Volumes

In the intersection testing stage of a ray tracing algorithm, a ray is tested against all the objects in a scene to find out exactly which object is hit by the ray. Because 3D objects may have many different shapes, it is usually very difficult and time-consuming to check the intersection between the ray and objects. Therefore, researchers have introduced several simpler bounding volumes[20][23], such as spheres, boxes, cylinders, to enclose these real objects with different 3D shapes. After all the objects in the scene are bounded by one type of bounding volume, e.g., spheres, the difficulties of intersection checks between the ray and many objects with different shapes is reduced and the gain in efficiency is significant because their shapes are unified.

Convexity is a geometrical property usually desired for bounding volumes because it guarantees that any ray will intersect the volume at most twice. Spheres, boxes and cylinders are all convex shapes and valid to serve as bounding volumes.

Whitted[21] initially used spheres as bounding volumes, observing that they are the simplest shapes to test for intersection.

Two advantages of the sphere as a bounding volume are:

- It is easy to find the sphere's center and radius.
- The number of calculations for testing the ray/sphere intersection is the smallest.

For example, when spheres are selected as bounding volumes in the new algorithm

developed in the thesis (Chapter 5), only the centers of the spheres need to be translated and rotated in order to determine the new locations of the objects. However, if boxes are selected, the same set of calculations has to be applied to 8 vertices of the box before an object's new position could be known. So, using spheres as the bounding volumes is about eight times faster in the new algorithm than using boxes.

3.3 Hierarchical Bounding Volumes(HBV)

The use of bounding volumes make the intersection testing operations simpler than the original shapes. However, it does not actually decrease the number of primitive objects in the scene. According to the theoretical analysis, this might reduce the computational complexity by a constant factor. In order to improve this case, Rubin and Whitted[11] used hierarchical bounding volumes instead of separate volumes. This technique obtains a theoretical logarithmic time complexity in the number of objects. The idea is to enclose many smaller bounding volumes (*sibling volumes*) within a bigger one (*parent volume*). If a ray did not intersect the parent volume, it is not needed to conduct intersection testing against the sibling volumes within. A hierarchy should be constructed before real intersection testing operations.

Rubin and Whitted employed oriented bounding boxes as bounding volumes to minimize void area. Before the ray/box intersection testing operations, the ray is first transformed into the defined coordinate system of the bounding box. Testing the

```
Procedure    BVH_Intersect( ray, node)
begin
    if    node is a leaf    then
        Intersect(ray, node.object)
    else if    Intersect_B(ray, node.bounding_volume)    then
        for    each child of node    do
            BVH_Intersect(ray, child);
end
```

Figure 3.1: A procedure for intersecting a ray with a collection of objects organized in a bounding volume hierarchy. Procedure 'Intersect' and function 'Intersect_B' hide many of the low-level details.

intersection between the transformed ray and axis-aligned bounding box becomes easier because the coordinate system was always the same; it is aligned to the box now. Since it is very simple to conduct the ray transformation operation, the bounding boxes are also used for representing the objects at the terminal nodes of a hierarchy. Rubin and Whitted also tried to simplify the polygonal representations by one or more bounding boxes along one axis.

3.4 An Algorithm With HBV

Figure 3.1 shows a procedure, namely '**BVH_Intersect**', that intersects a ray with many objects organized in a bounding volume hierarchy. The data structure of this hierarchy is similar to a tree with an arbitrary branching factor at each internal node. Thus, parent bounding volumes may enclose any number of other sibling bounding

volumes. Each leaf node of the tree is a single primitive and the interior node is a bounding volume with a list of pointers to its sibling nodes.

The procedure '**Intersect**' in '**BVH_Intersect**' is responsible for calling the appropriate ray-object intersection operation for this type of primitive object. The '**ray**' parameter includes a 3-D origin, a direction vector, and a **distance interval**. Points of intersection farther than the distance interval (measured along the ray from its origin) are to be ignored and a newly found point of intersection in '**Intersect**' always replaces the far end of the distance interval to that point.

The procedure '**Intersect_B**' is very similar to '**Intersect**' except that it returns a boolean value indicating whether an intersection might happen and it does *not* actually alter the ray's distance interval. This function is used to test if the particular bounding volume is hit by the ray.

With '**Intersect**' and '**Intersect_B**', the work of intersecting a ray within a certain bounding volume hierarchy is quite straightforward. The process begins with the root node of the tree, representing a bounding volume enclosing the entire environment, and the ray is assigned the distance interval equal to 'infinity' before real processing. Each recursive reference of '**BVH_Intersect**' descends another level of the hierarchy, and the recursion terminates with real ray-object intersection tests at the leaves. At each level, the ray is tested against all the sibling bounding volumes and only descends into the ones which are hit by the ray. Those sibling volumes not hit by the ray are

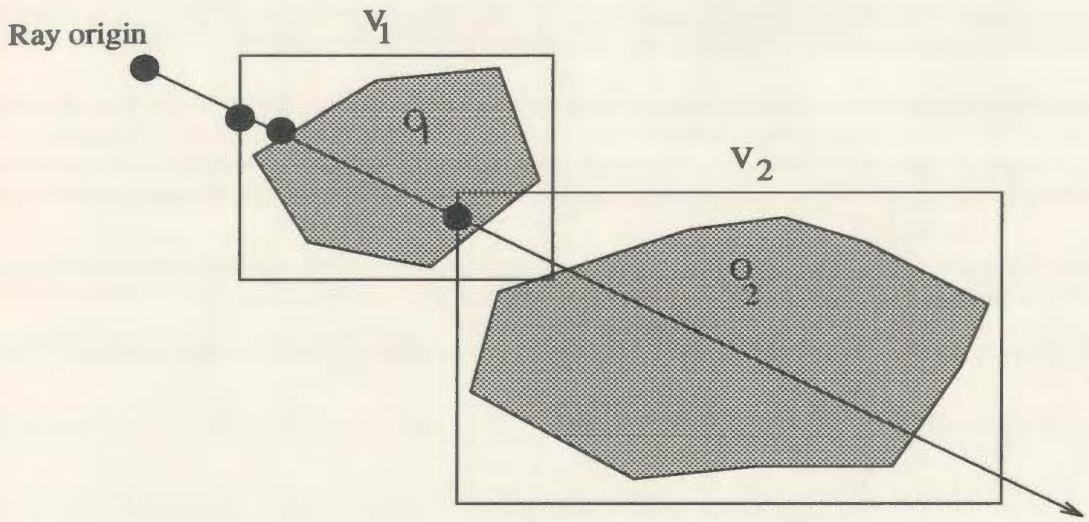


Figure 3.2: An optimization which results from shrinking the distance interval associated with a ray whenever an intersection is found. The contents of volume V_2 need not be tested against this ray if the intersection with object O_1 is found first.

ignored and the *prune* function is realized.

The benefit of adjusting the ray's distance interval is that it performs a useful optimization [4, 10]. Once an intersection point has been found with some object, or bounding volume with an upper bound of the distance interval, all objects or bounding volumes farther than this bound do not need be tested. This provides a second prune mechanism from the hierarchy during the processing of a ray. An example of this is shown in Figure 3.2. If bounding volume V_1 is processed before V_2, the contents of the latter need not be tested because the point of intersection with object O_1 is closer than any within V_2. This saves at least one ray-object intersection test and potentially many in cases where V_2 encloses other sibling bounding volumes.

3.5 Theoretical Analysis of Bounding Volume Optimization

In section 3.2, it is already observed (Whitted[21]) that the sphere is the simplest shape to conduct intersection calculations. However, people still do not know if the sphere is the best bounding volume and what is the relationship between bounding volume and the cost of intersection. This section addresses this question.

Weghorst *et al.* [16] investigated this by considering the two major factors of this problem: tightness of fit and the cost of intersection. It was found that the total computational cost associated with an object and its bounding volume could be expressed by

$$Cost = n * B + m * I \tag{3.1}$$

That means the total cost has two components, the cost (B) spent for conducting intersection testing between all rays (n) with the object's bounding volume and the cost (I) spent for conducting intersection testing between some rays (m) with this primitive object. n is the number of rays tested against the bounding volume. m is the number of rays which actually hit the volume. Since n, I, and B are all relatively fixed, it is better to select a bounding volume which minimizes m , i.e., as tight fitting as possible because the smaller the bounding volume is, the smaller the number of

rays that will hit it. Now, it is necessary to set a standard for measuring the fit of the bounding volumes. Weghorst *et al.* used the enclosed volume as a measure of fit, observing that it is related to the *projected void area* with respect to any direction, i.e., to the difference in the projected areas of the bounding volume and the enclosed objects. This difference in area indicates how likely a ray is to hit the bounding volume without hitting the enclosed object. A large void area, resulting from a loose fit, can increase m and cause many unnecessary object intersection checks. Reducing m even at the expense of an increase in B is sometimes warranted. Weghorst *et al.* introduced a simple method to determine when such a trade-off is likely to be advantageous.

1. Each type of bounding volume, such as a sphere, box, cylinder etc., was assigned a relative complexity factor to rank the computational cost of the ray intersection tests. In their implementation, spheres were given the lowest complexity rating and cylinders the highest.

2. Each volume was 'tried' in turn as a potential bound, and the one producing the smallest *product* of volume and complexity factor was selected. This applies equally well to the bounding volumes of the internal nodes of a hierarchy. Because this method did not take the complexity of the enclosed object into account, however, an interactive program was used to occasionally override the algorithmically selected bounding volume.

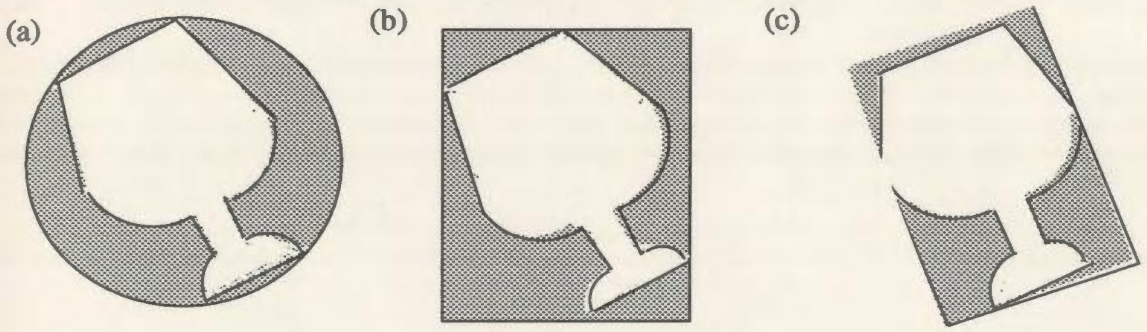


Figure 3.3: A comparison of three different types of bounding volumes for the same primitive object. Each presents a different cost/fit ratio. (a) Bounding sphere. (b) Axis-aligned bounding box. (c) Oriented bounding box.

Figure 3.3 shows three possible bounding volumes for a complex object and the shaded region represents the projected void area. This void area usually depends upon the direction along which the two-dimensional projection is formed. Since the rays in the environment are effectively randomized by multiple reflections and refraction, the average projected void area (over all directions) becomes the relevant measure of fit.

Volumes(b) and (c) in Figure 3.3 are axis-aligned and transformed (oriented) bounding boxes, respectively. The latter clearly produces a better fit but needs the extra cost of a ray transformation for every ray-bounding volume intersection check.

3.6 Approximate Convex Hulls

A *convex hull* is a uniquely defined bounding volume and has the property of most tightness for a certain bounded object. Therefore, it may be an exemplary bounding volume according to the standard mentioned in the last section. However, the computation and representation of the convex hull can be difficult. Therefore, an approximation of the true convex hull may have to be used so that the resulting volume can easily be used to conduct the intersection operations, otherwise the convex hull is useless even though it might have the least void area.

Kay and Kajiya[9] designed one type of approximate convex hull, i.e. the many-sided parallelepiped, which can be made as close to the object as the actual convex hull. The algorithm uses the concept of *plane-sets* which are actually families of parallel planes. Each plane-set is defined according to a single unit vector, *the plane-set normal*, and each plane of a family is uniquely determined by its signed distance from the origin (equal to the inner product of the plane-set normal and any point on the plane). Given a plane-set normal and an arbitrary (bounded) object, there exist two such planes which most closely bracket the object. The infinite region between the two planes is called a *slab*, and is conveniently represented by a min-max interval associated with the plane-set normal as shown in Figure 3.4(a).

The intersection of several different slabs can define a bounded region totally enclosing the object, as shown in Figure 3.4(b). In 3-D space, three slabs whose plane-set

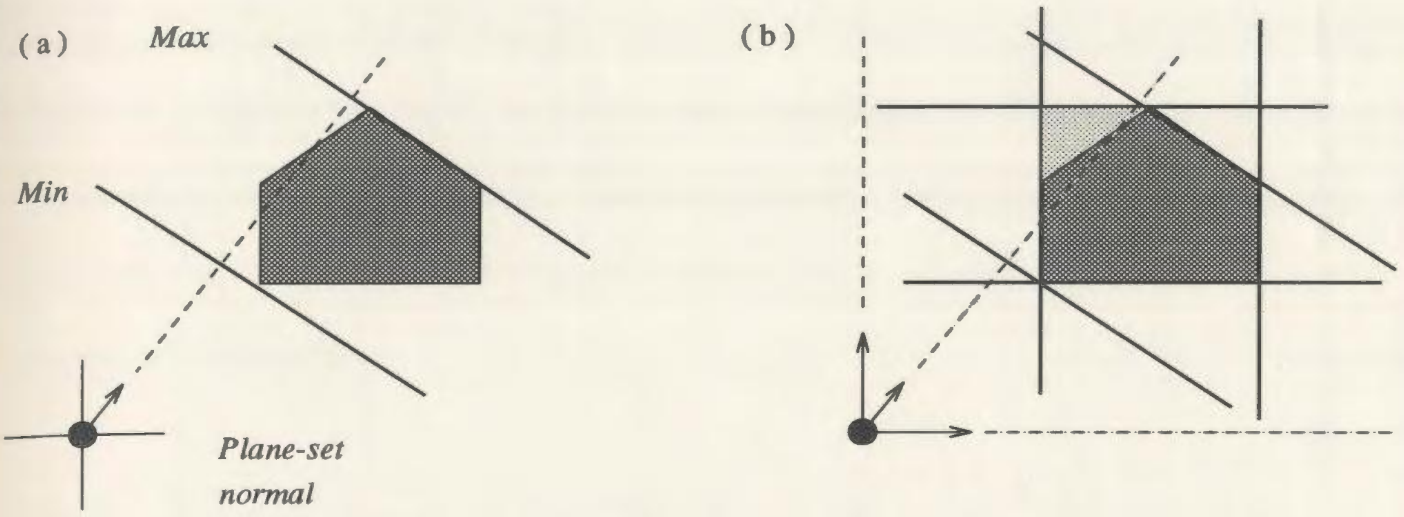


Figure 3.4: A plane-set normal defines a family of parallel planes orthogonal to it. Two values associated with a plane-set normal select two of these planes and define a slab. The intersection of several such slabs forms a parallelepiped bounding volume. (a) A single slab bracketing an object. (b) Three slabs defining a bounding volume.

normals are linearly independent are necessary (two suffice in two-space), but the number of slabs is by no means limited to three. The greater the number of slabs, the more closely the actual convex hull of the object can be approximated. To intersect a ray with such a volume the interval along the ray is first computed, measured from its origin, which lies within each of the slabs. This amounts to computing two ray-plane intersections for each slab. If the intersection of these intervals is empty, the ray misses the volume. Otherwise, the ray hits the volume and the maximum of the minimum interval values is the distance to the point of intersection.

It is better to use the same collection of plane-set normals for all the objects in the environment, despite their individual orientations. The most significant advantage is that the task of intersecting a ray with a number of bounding volumes can be

greatly accelerated because certain rays just need to be transformed once and common expressions in the ray-plane intersection calculations can be 'factored out' and done once per ray instead of once per bounding volume. The calculations required here are only two subtractions, two multiplications and a comparison for each slab of a bounding volume[9].

3.7 Predicting the Effectiveness of a Hierarchy

The effectiveness of a bounding volume also depends on the distribution of rays which will be tested against it. If every ray were to hit the enclosed object, no bounding volume would be useful because every type of bounding volume, no matter how simple, would only increase the cost of the intersection checks. On the other hand, if no ray even approaches the enclosed object, any type of bounding volumes with less cost to test than the object is better. In most situations the distribution of rays falls somewhere between these two extremes.

It is time to study how ray distributions affect the BVH because it can be used to predict the performance of a BVH. Goldsmith and Salmon[6] in particular study the conditional probability of a ray hitting a sibling volume, B, provided that it has hit the parent volume, A. See Figure 3.5(a). This conditional probability can be expressed by $P_r(r \text{ hits } B \mid r \text{ hits } A)$, where r is a 'random' ray and all the rays hit A are assumed to be uniformly distributed. It is clear that A helps to filter out those rays which would

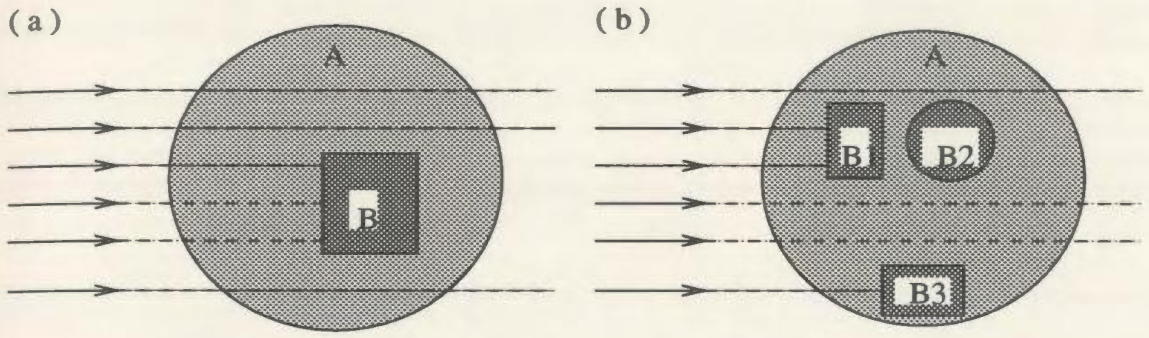


Figure 3.5: (a) An approach of computing the conditional probability of a ray hitting B given that it has hit A. This can be used in cases like (b) to compute the average cost of intersecting a ray with the arbitrary contents of a bounding volume.

not hit B.

It was found that $P_r(r \text{ hits } B \mid r \text{ hits } A)$ is equal to the ratio of the average projected area of B to the average projected area of A. This discovery makes the ray distribution computable because it is already known that the average projected area of a convex body is equal to one quarter of its surface area [14, p.110]. Since A and B are convex bounding volumes, the ray distribution P_r can be calculated by the formula,

$$Pr(r \textit{ hits } B \mid r \textit{ hits } A) = \frac{< P(B, \mathbf{d}) >}{< P(A, \mathbf{d}) >} = \frac{S(B)}{S(A)} \tag{3.2}$$

where $P(V, \mathbf{d})$ is the projected area of V (A or B) along direction $\mathbf{d}$, <> means the average taken over all directions $\mathbf{d}$, and $S(V)$ ($S(A)$ or $S(B)$) is the surface area of volume V. This formula is the basic tool to study the cost effect of a HBV.

There are usually two types of costs for a bounding volume:

1. *external cost* (EC) : the cost for ray/volume intersection operation.

2. *internal cost* (IC) : the average cost between the ray and the contents within the volume if the ray hits it.

Here, EC is actually the B and IC the I in Eq. 3.1. Now, since the conditional probability (Eq. 3.2) has been discovered, the IC in volume A can be computed by the formula:

$$IC(A) = \sum_{i=1}^{n} \left\{ EC(B_i) + \frac{S(B_i)}{S(A)} * IC(B_i) \right\} \tag{3.3}$$

where $EC(B_i)$ is the fixed cost of testing a ray against the ith individual sibling. $IC(B_i)$ is the internal costs of the siblings. $\frac{S(B_i)}{S(A)}$ are the different conditional probabilities for A's enclosing sibling volumes $B_1, B_2, \cdots, B_n$. The IC of a primitive is defined here to be zero. Thus, the average cost of intersecting a ray with a BVH is obtained by recursive application of Eq. 3.3 in terms of surfaces areas and ECs.

However, It should be noted that Eq. 3.3 is only an approximate formula because many assumptions were made. In addition to the proper nesting, convexity, and randomness assumptions noted earlier, an implicit assumption has been that the external cost (EC) of a bounding volume is constant for all rays and independent of whether or not the volume is hit by the ray. The effects of objects occluding one another are also neglected. For example, in Figure 3.5(b) any of the rays shown which hit an object within B_1 need not be tested against the contents of B_2 due to the distance interval

optimization.

3.8 Constructing A Hierarchy

Two types of decisions need to be made before constructing a BVH:

1. Which ones should be the enclosed objects or sibling bounding volumes.

2. What is the bounding volume selected.

The formulas and standards described in Section 3.5 and 3.7 are usually employed by such decisions.

However, since the number of possible hierarchy groupings of objects grows exponentially with the number of objects while constructing a BVH using the selected volume, it is actually impractical to make a exhaustive search. Rubin and Whitted[11] first solved this problem by using a *structure editor*, an interactive program which constructs successive levels of a BVH by beginning with those unstructured primitives. Thus, user can select group of objects according to their real space coherence and select proper tight-fitting bounding boxes for them as well. A means of performing this operation automatically was also suggested in [11].

Weghorst *et al.*[16] suggested that modeling hierarchies used in constructing the environment are often adequate for a ray tracing algorithm. The model builder usually

groups those objects in close proximity and this practice could reduce the average projected void area of the resulting bounding volume. However, Goldsmith and Salmon[6] pointed out that such hierarchies tend to have large branching factors, thereby reducing the benefits of tree pruning during ray intersection testing. In order to overcome this problem, they developed an algorithm for automatic generation of bounding volume hierarchies according to Eq. 3.3. In the method, the hierarchy is constructed incrementally, inserting the primitives into the growing structure one at a time while striving to minimize the resulting increases of the bounding volume surface areas. Each primitive starts at the root of the tree and selects the subtree which would incur the smallest increase of surface areas if the new object were to become a child of it. This process continues until reaching a leaf of the tree.

Goldsmith and Salmon observed that the order of the inserted objects into the BVH is very important for the eventual form of the tree. The order could actually have three kinds, i.e. the imposed model order, the sorted order along a line and randomized. It was discovered that the best trees would be constructed by trying many different randomized orders.

Chapter 4

Two Current Ray/Sphere Intersection Testing Techniques

The preceding survey suggests that spheres are the simplest and most practical bounding volumes and therefore are widely used in ray tracing [27]. Now, we shall examine current algorithms which perform ray/sphere intersection testing task and see if we can improve their efficiency.

Until now, there are two principal ray/sphere intersection testing algorithms, the traditional algebraic solution and a more efficient geometric solution[18]. The two techniques are introduced in detail in this chapter.

4.1 Algebraic Solution

The intersection point between a ray and a sphere can be obtained algebraically from their mathematical definition[29]. Let $\mathbf{r}(t)$ be the ray and $\mathbf{s}(x, y, z)$ the sphere. Given the origin $\mathbf{r}_o = [x_o\ y_o\ z_o]^T$ and direction $\mathbf{r}_d = [l\ m\ n]^T$, $\mathbf{r}(t)$ is defined by $\mathbf{r}(t) = \mathbf{r}_o + \mathbf{r}_d t$, where t is a parameter and $t > 0$. The equation can also be written as follows:

$$\begin{aligned} x(t) &= x_o + lt \\ y(t) &= y_o + mt \qquad t > 0 \\ z(t) &= z_o + nt \end{aligned} \tag{4.1}$$

$\mathbf{s}(x, y, z)$ is defined implicitly with its center $\mathbf{x}_c = [x_c\ y_c\ z_c]^T$ and radius r_s.

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r_s^2 \tag{4.2}$$

Substituting x, y, and z in Eq. 4.2 with $x(t), y(t)$, and $z(t)$ from Eq. 4.1 yields the following relation,

$$(x_o + lt - x_c)^2 + (y_o + mt - y_c)^2 + (z_o + nt - z_c)^2 = r_s^2 \tag{4.3}$$

which can be rewritten as,

$$At^2 + Bt + C = 0 \tag{4.4}$$

where

$$A = l^2 + m^2 + n^2 \tag{4.5}$$

$$B = 2l(x_0 - x_c) + 2m(y_0 - y_c) + 2n(z_0 - z_c) \tag{4.6}$$

$$C = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r_s^2 \tag{4.7}$$

When the ray's direction $\mathbf{r}_d$ is a unit vector, $A = 1$. Therefore t can be obtained using the quadratic formula, Eq. 4.4.

$$t_{0,1} = \frac{-B \pm \sqrt{(B^2 - 4C)}}{2} \tag{4.8}$$

Since $t > 0$, the ray intersects with the sphere only if there is a positive real root t_i, in Eq. 4.8, $i = 0$ or 1. If there are two such roots, the smaller one is chosen as it stands for the closer intersection to $\mathbf{r}_o$.

Once the distance t is found, the actual intersection point is:

$$\mathbf{r}_{intersect} \equiv \mathbf{r}_i = [x_i \; y_i \; z_i] = [x_o + l * t, \; y_o + m * t, \; z_o + n * t] \tag{4.9}$$

The unit vector normal at the surface is then simply:

$$\mathbf{r}_{normal} \equiv \mathbf{r}_n = \left[\begin{array}{ccc} \frac{x_i - x_c}{r_s} & \frac{y_i - y_c}{r_s} & \frac{z_i - z_c}{r_s} \end{array} \right] \tag{4.10}$$

To summarize, the steps in the algorithm are:

Step 1: calculation of A, B, and C of the quadratic.

Step 2: calculation of discrimination.

Step 3: calculation of t_0 and comparison.

Step 4: possible calculation of t_1 and comparison.

Step 5: intersection point calculation.

Step 6: calculation of normal at point.

The calculations associated with each step are:

Step 1: 8 additions/subtractions and 7 multiplies.

Step 2: 1 subtraction, 2 multiplies, and 1 compare.

Step 3: 1 subtraction, 1 multiply, 1 square root, and 1 compare.

Step 4: 1 subtraction, 1 multiply, and 1 compare.

Step 5: 3 additions, 3 multiplies.

Step 6: 3 subtractions, 3 multiplies.

For the worst case this gives a total of 17 additions/subtractions, 17 multiplies, 1 square root, and 3 compares.

4.2 Geometric Solution

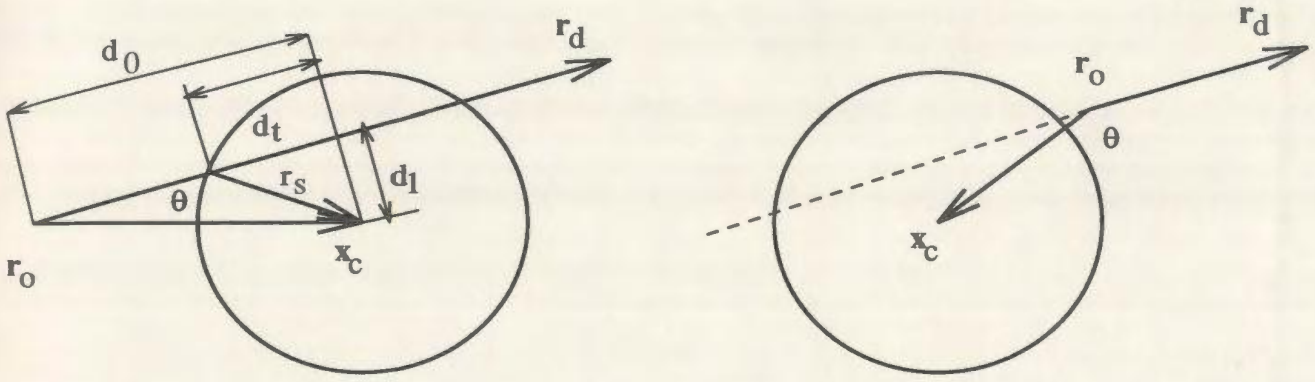
With the simple sphere intersection routine being outlined, the next question is, "How can we make it run faster ?" After studying the geometry of the ray/sphere situation, Haines[18] points out that there are a number of small tests which can be made to determine whether an ray/sphere intersection takes place. The purpose of these small tests is to avoid ray/sphere intersection calculations until they are needed.

Actually, the intersection between a ray and a sphere depends completely on the spatial relation[30] between the ray ($\mathbf{r}_o$ and $\mathbf{r}_d$) and the sphere ($\mathbf{x}_c$ and r_s). A ray whose origin is outside of a sphere will never hit the sphere if the ray points away from it. As Eq. 4.2 defines the set of surface points on the sphere, the origin of a ray is inside of the sphere only if the distance between $\mathbf{r}_o$ and $\mathbf{x}_c$ is smaller than r_s.

$$\sqrt{(x_o - x_c)^2 + (y_o - y_c)^2 + (z_o - z_c)^2} < r_s \tag{4.11}$$

Squaring both sides leads to the following condition.

$$(x_o - x_c)^2 + (y_o - y_c)^2 + (z_o - z_c)^2 < r_s^2 \tag{4.12}$$



(a) to a sphere (b) away from a sphere

Figure 4.1: The Geometric Relation between a Ray and a Sphere

Meanwhile, the sign of the dot product d_0 between $\mathbf{r}_d$ and vector $\mathbf{r}_o\vec{\mathbf{x}}_c$ tells if a ray points away from a sphere when the ray is outside of the sphere. This dot product

$$\begin{aligned} d_0 &= \mathbf{r}_d \cdot \mathbf{r}_o\vec{\mathbf{x}}_c = |\mathbf{r}_d||\mathbf{r}_o\vec{\mathbf{x}}_c|cos\theta \\ &= l(x_c - x_o) + m(y_c - y_o) + n(z_c - z_o) \end{aligned} \tag{4.13}$$

is positive if the ray points to the sphere (Figure 4.1(a)) or negative when pointing away (Figure 4.1(b)).

In addition, the squared distance d_1^2 between the ray and the sphere center $\mathbf{x}_c$ can be obtained from d_0 and $|\mathbf{r}_o\vec{\mathbf{x}}_c|$ (Figure 4.1(a)).

$$d_1^2 = |\mathbf{r}_o\vec{\mathbf{x}}_c|^2 - d_0^2 \tag{4.14}$$

It is also a useful measurement for testing for intersection, i.e., a ray will miss the sphere if $d_1^2 > r_s^2$ when it starts outside of and points towards the sphere.

A ray that cannot be eliminated by d_0 and d_1 must intersect the sphere. Since d_t in Figure 4.1(a) is the square root of r_s^2 minus d_1^2,

$$d_t = \sqrt{r_s^2 - d_1^2} \tag{4.15}$$

the actual distance from $\mathbf{r}_o$ to the intersection point is $d_0 - d_t$ if $\mathbf{r}_o$ is outside of the sphere, or $d_0 + d_t$ if $\mathbf{r}_o$ is on or inside of the sphere. The spheres that definitely are hit by the ray are sorted by distance, for actual surface intersection. This function is provided in the previous section by the algebraic solution of Eq. 4.8.

The testing and elimination procedures enable a ray tracing algorithm to avoid computing the real surface interactions unless it is absolutely necessary. Based on the geometric relations, an improved algorithm was generated[18], which has been recognized for its efficiency in ray tracing [31, 32].

To summarize, the steps in the algorithm are:

Step 1: find distance squared between ray origin and center.

Step 2: calculate ray distance which is closest to center.

Step 3: test if ray is outside and points away from sphere.

Step 4: find square of half chord intersection distance.

Step 5: test if square is negative.

Step 6: calculate intersection distance.

Step 7: find intersection point.

Step 8: calculate normal at point.

The calculations associated with each step are:

Step 1: 5 additions/subtractions and 3 multiplies.

Step 2: 2 additions and 3 multiplies.

Step 3: 2 compares(1 if origin inside sphere).

Step 4: 2 additions/subtractions and 1 multiply.

Step 5: 1 compare (none if origin inside sphere).

Step 6: 1 addition/subtraction and 1 square root.

Step 7: 3 additions, 3 multiplies.

Step 8: 3 subtractions, 3 multiplies.

At worst this gives a total of 16 additions/subtractions, 13 multiplies, 1 square root, and 3 compares. Note that this is less than the algebraic solution.

Chapter 5

A New Algorithm

Although the use of spheres as bounding volumes delays computing the complicated ray-surface intersection, the testing of ray/sphere intersection still takes too much time. This problem arises because of the three-dimensional relationship between ray and spheres as all of them are given in the three-dimensional definition space — the world coordinate system. If, however, a plane is perpendicularly attached to the ray and all spheres are projected onto the plane as circles, only those spheres for which the ray goes through their projected circles can intersect with the ray. As a result, the three-dimensional ray/sphere interaction is simplified into a two-dimensional point-circle enclosure check. This is the observation that promoted the new algorithm.

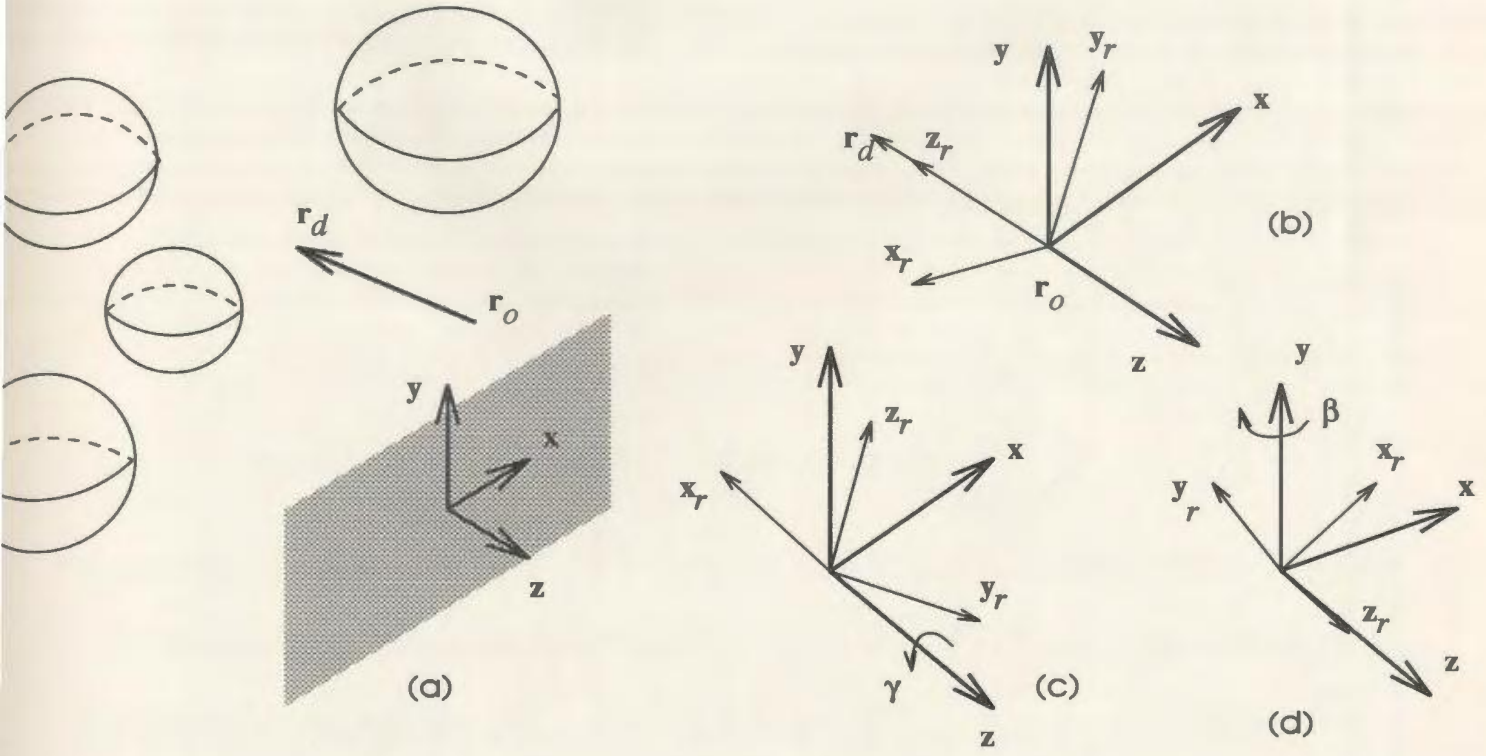


Figure 5.1: The Definition and Transform of the Ray System

5.1 From Intersection to Enclosure Check

The point-circle enclosure check is conducted after transforming the spheres to a coordinate system defined for the ray. This process includes the following steps:

1. Define a ray coordinate system $F_r = \{(x_r, y_r, z_r)\}$ and make $\mathbf{r}_d$ the axis $\mathbf{z}_r$.

2. Obtain the matrix that transforms the coordinates in the world coordinate system $F_w = \{(x_r, y_r, z_r)\}$ into the coordinates in $\{F_r\}$.

3. Apply the matrix to all the spheres to obtain their definition in the ray coordinate system.

4. Discard those spheres that are totally in the negative $\mathbf{z}_r$ half space of $\{F_r\}$.

5. Project the remaining spheres onto the $\mathbf{x}_r - \mathbf{y}_r$ plane.

6. Find and save the spheres whose projected circles enclose the origin of the $\mathbf{x}_r - \mathbf{y}_r$ plane.

5.1.1 Coordinate System Transformation

The ray coordinate system $\{F_r\}$ is defined in such a way that its origin coincides with $\mathbf{r}_o$ and its $\mathbf{z}_r$ axis points in the same direction as $\mathbf{r}_d$ (Figure 5.1(b)). The matrix that transforms coordinates $\{(x, y, z)\}$ into coordinates $\{(x_r, y_r, z_r)\}$ is obtained from the homogeneous matrices that specify a series of transformations bringing the $\mathbf{z}_r$ axis of $\{F_r\}$ to the $\mathbf{z}$ axis of $\{F_w\}$[33].

To bring $\{F_r\}$ to $\{F_w\}$, the origin of $\{F_r\}$ is first shifted to the origin of $\{F_w\}$ by a homogeneous translation matrix $T(-\mathbf{r}_o)$ (Figure 5.1(b)).

$$T(-\mathbf{r}_o) = \begin{bmatrix} 1 & 0 & 0 & -x_o \\ 0 & 1 & 0 & -y_o \\ 0 & 0 & 1 & -z_o \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

The shifted $\{F_r\}$ is then rotated for an angle γ, $0 \leq \gamma \leq 360^0$, around the $\mathbf{z}$ axis of $\{F_w\}$ to lay axis $\mathbf{z}_r$ onto the $\mathbf{z} - \mathbf{x}$ plane with $\mathbf{z}_r$ pointing in the positive direction of

$\mathbf{x}$ (Figure 5.1(c)).

$$R_z(\gamma) = \begin{bmatrix} cos\gamma & -sin\gamma & 0 & 0 \\ sin\gamma & cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

The $\mathbf{z}_r$ axis is finally brought to coincide with the $\mathbf{z}$ axis by the last rotation $R_y(\beta)$, which is a rotation around the $\mathbf{y}$ axis of $\{F_w\}$ for an angle β, $-180^0 \leq \beta \leq 0^0$ (Figure 5.1(d)).

$$R_y(\beta) = \begin{bmatrix} cos\beta & 0 & sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\beta & 0 & cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

As a result, the transformation matrix τ that transforms coordinates $\{(x, y, z)\}$ into coordinates $\{(x_r, y_r, z_r)\}$ is the product of matrices $T(-\mathbf{r}_o)$, $R_z(\gamma)$, and $R_y(\beta)$,

$$\tau = R_y(\beta)R_z(\gamma)T(-\mathbf{r}_o)$$

$$
\begin{aligned}
&= \begin{bmatrix} c\beta & 0 & s\beta & 0 \\ 0 & 1 & 0 & 0 \\ -s\beta & 0 & c\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c\gamma & -s\gamma & 0 & 0 \\ s\gamma & c\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_o \\ 0 & 1 & 0 & -y_o \\ 0 & 0 & 1 & -z_o \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} c\beta c\gamma & -c\beta s\gamma & s\beta & 0 \\ s\gamma & c\gamma & 0 & 0 \\ -s\beta c\gamma & s\beta s\gamma & c\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_o \\ 0 & 1 & 0 & -y_o \\ 0 & 0 & 1 & -z_o \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} c\beta c\gamma & -c\beta s\gamma & s\beta & -x_o c\beta c\gamma + y_o c\beta s\gamma - z_o s\beta \\ s\gamma & c\gamma & 0 & -x_o s\gamma - y_o c\gamma \\ -s\beta c\gamma & s\beta s\gamma & c\beta & x_o s\beta c\gamma - y_o s\beta s\gamma - z_o c\beta \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{5.4}
$$

where $c\gamma = cos\gamma, s\gamma = sin\gamma, c\beta = cos\beta, s\beta = sin\beta$. Since $\mathbf{r}_d$ points to the direction $[0\ 0\ 1]^T$ after the transformation, they are related by the rotational component of τ[34].

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} c\beta c\gamma & -c\beta s\gamma & s\beta & 0 \\ s\gamma & c\gamma & 0 & 0 \\ -s\beta c\gamma & s\beta s\gamma & c\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} l \\ m \\ n \\ 1 \end{bmatrix} \tag{5.5}$$

where l, m, and n are the three components of $\mathbf{r}_d$. An inverse of the transformation matrix relates l, m, and n with $c\gamma, s\gamma, c\beta$, and $s\beta$.

$$\begin{bmatrix} l \\ m \\ n \\ 1 \end{bmatrix} = \begin{bmatrix} c\beta c\gamma & -c\beta s\gamma & s\beta & 0 \\ s\gamma & c\gamma & 0 & 0 \\ -s\beta c\gamma & s\beta s\gamma & c\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} c\beta c\gamma & s\gamma & -s\beta c\gamma & 0 \\ -c\beta s\gamma & c\gamma & s\beta s\gamma & 0 \\ s\beta & 0 & c\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

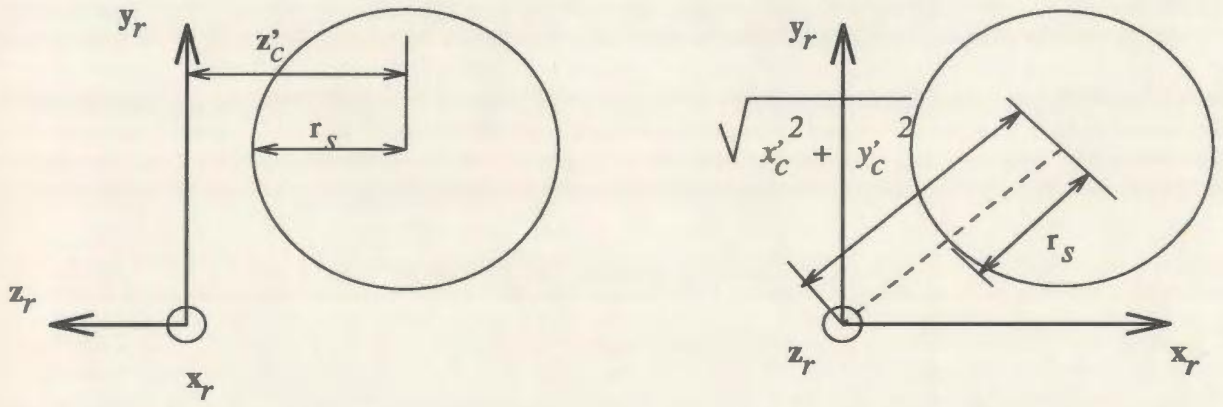
$$= \begin{bmatrix} -s\beta c\gamma \\ s\beta s\gamma \\ c\beta \\ 1 \end{bmatrix} \tag{5.6}$$

As a consequence, $c\gamma, s\gamma, c\beta$, and $s\beta$ are derived from Eq. 5.6, where the minus sign of $s\beta$ is selected due to the angle of β, $-180^0 \leq \beta \leq 0^0$.

$$\begin{aligned} s\beta &= -\sqrt{1 - c\beta^2} = -\sqrt{1 - n^2} \\ c\beta &= n \\ s\gamma &= \frac{m}{s\beta} = -\frac{m}{\sqrt{1 - n^2}} \\ c\gamma &= -\frac{l}{s\beta} = \frac{l}{\sqrt{1 - n^2}} \end{aligned} \tag{5.7}$$

By substituting Eq. 5.7 into Eq. 5.4, the transformation matrix τ is finally obtained.

$$\tau = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



(a) a sphere behind (b) a sphere out of range

Figure 5.2: The Point-Circle Enclosure Check

$$= \begin{bmatrix} \frac{ln}{\sqrt{1-n^2}} & \frac{mn}{\sqrt{1-n^2}} & -\sqrt{1-n^2} & -x_o\frac{ln}{\sqrt{1-n^2}} - y_o\frac{mn}{\sqrt{1-n^2}} + z_o\sqrt{1-n^2} \\ -\frac{m}{\sqrt{1-n^2}} & \frac{l}{\sqrt{1-n^2}} & 0 & x_o\frac{m}{\sqrt{1-n^2}} - y_o\frac{l}{\sqrt{1-n^2}} \\ l & m & n & -x_o l - y_o m - z_o n \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.8)$$

5.1.2 Point-Circle Enclosure Check

The new algorithm is outlined below, which is a two-dimensional point-circle enclosure check instead of the three-dimensional ray/sphere intersection.

step 0: Obtain the sphere-list and create a hit-list.

step 1: Use Eq. 5.8 to compute the homogeneous matrix τ from ray's origin $\mathbf{r}_o$ and direction $\mathbf{r}_d$.

step 2: Take one sphere from the sphere-list and calculate the $z_c^{'}$ component of the new origin $\mathbf{x}_c$ in $\{F_r\}$.

$$z_c^{'} = m_{20}x_c + m_{21}y_c + m_{22}z_c + m_{23} \tag{5.9}$$

step 3: If $z_c^{'} < -r_s$, the sphere must be behind the ray (Figure 5.2(a)), i.e., no intersection takes place between the ray and the sphere. Discard the sphere from the sphere-list and go back to **step 2** to continue processing the other spheres.

step 4: Otherwise, compute the $x_c^{'}$ and $y_c^{'}$ component of origin $\mathbf{x}_c$.

$$x_c^{'} = m_{00}x_c + m_{01}y_c + m_{02}z_c + m_{03} \tag{5.10}$$

$$y_c^{'} = m_{10}x_c + m_{11}y_c + m_{13} \tag{5.11}$$

step 5: Calculate $x_c^{'2} + y_c^{'2}$, If $x_c^{'2} + y_c^{'2} > {r_s}^2$, the ray misses the sphere as it passes outside of the sphere (Figure 5.2(b)). Discard the sphere and go back to **step 2**.

step 6: Remove the sphere from the sphere-list, and add it to the hit-list in a position determined by its $z_c^{'}$ value.

step 7: If the sphere-list is not empty, go back to step **2**. Otherwise, the check is finished.

Chapter 6

Complexity and Comparison

From the discussion of Chapter 4 and 5, three ideas are clear:

1. There are currently two main ray/sphere intersection algorithms, the algebraic solution and the geometric solution, and the geometric solution is the faster of the two. If, in the worst case, the new algorithm is faster than the geometric solution, it will also be faster than the algebraic solution.

2. After careful analysis, the algebraic solution actually can be divided into two parts: first, conduct some calculations in order to check whether or not the current ray hits the sphere, (Step 1 to Step 4 in Section 4.1); we can call this stage the *pre-determination* part. Second, calculate the intersection point coordinates and normal values, (Step 5,6 in Section 4.1) we call this stage the *evaluation* part. It is also noticed that the geometric solution only makes some improvements

(Step 1 to Step 6 in Section 4.2) to the *pre-determination* part of the algebraic algorithm and keeps the second *evaluation* part unchanged.

3. The new algorithm also makes some improvements to the *pre-determination* stage by employing homogeneous transformations operations to realize 3D to 2D simplification.

After understanding the above three points, it is clear that the complexity comparison of this Chapter should only be conducted between geometric solution and the new algorithm. It is also clear that only comparison of their *pre-determination* stages is necessary.

In order to facilitate the complexity analysis and comparison, we restate the steps of the *pre-determination* part of the geometric solution of ray/sphere intersection algorithms in Section 4.2 as follows:

1. Calculate the squared distance $|\mathbf{r}_o\vec{\mathbf{x}}_c|^2$ between $\mathbf{r}_o$ and $\mathbf{x}_c$.

2. Calculate the dot product d_0 of $\mathbf{r}_d$ and $\mathbf{r}_o\vec{\mathbf{x}}_c$.

3. Use Eq. 4.10 and the sign of d_0 to check if the ray is outside of and points away from the sphere. If yes, go back to 1. to process the other spheres.

4. Otherwise, compute d_1^2 from Eq. 4.12. If $d_1^2 > r_s^2$, go back to 1.

5. Compute d_t from Eq. 4.13. If the ray's origin is outside of the sphere, use $d_0 - d_t$ and continue to 6.; otherwise, use $d_0 + d_t$.

6. Go back to 1, unless all spheres have been processed.

6.1 The Complexities

The computational complexity of an algorithm is determined by the number of operations necessary to implement the algorithm. In this section, the complexity of the new algorithm is analyzed first. Then, after obtaining the complexity of the improved geometric algorithm, the advantage of the new algorithm becomes apparent with a complexity comparison.

The number of operations is obtained from the algorithm of point-circle enclosure check introduced in Section 5.1.2 with **step 0** ignored because it is a common step of all raytracing algorithms.

- In **step 1**, the operations to calculate τ are *6* additions, *13* multiplications, and *1* square root (Table 6.1). While the operations in the step are needed only once for a given ray, in the remaining steps the number of spheres, supposedly N, has to be multiplied by the number of operations.

- To calculate z_c'(Eq. 5.9), **step 2** needs *3* additions and *3* multiplications.

order	operation	add/minus	multiply/divide	square root
a.	$n^2 = nn$		1	
b.	$1 - n^2$	1		
c.	$\sqrt{1-n^2}$			1
d.	$m_{10} = (-m)/\sqrt{1-n^2}$		1	
e.	$m_{11} = l/\sqrt{1-n^2}$		1	
f.	$m_{00} = nm_{11}$		1	
g.	$m_{01} = nm/\sqrt{1-n^2}$		1	
h.	$m_{02} = (-\sqrt{1-n^2})$			
i.	$m_{03} = -x_o m_{00} - y_o m_{01} + z_o m_{02}$	2	3	
j.	$m_{13} = x_o m_{10} - y_o m_{12}$	1	2	
k.	$m_{23} = (-x_o l) - y_o m - z_o n$	2	3	
total		6	13	1

Table 6.1: The Number of Operations to Calculate τ

- In **step 3**, *1* comparison is used to check the condition $z_c^{'} < -r_s$.
- Then, in **step 4**, $x_c^{'}$ requires *3* additions and *3* multiplications, and $y_c^{'}$, *2* additions and *2* multiplications.
- In the final checking stage **step 5**[1], *1* addition and *1* comparison are needed on top of the *2* multiplications created by $x_c^{'2}$ and $z_c^{'2}$.

The number of operations for the new algorithm is given as C_n in the following equation. For simplicity, symbols $\oplus$ (addition), $\otimes$ (multiplication), $\oslash$(comparison), and $\sqrt{\ }$ (square root) are used to stand for different types of operations. They are also used in the next section to denote basic time units of the operations.

[1]Since r_s^2 is not associated with any particular ray, it is computed outside of the algorithm.

$$
\begin{aligned}
C_n &= 6 \oplus +13 \otimes +1\surd + N(9 \oplus +10 \otimes +2\oslash) \\
&= (6+9N) \oplus +(13+10N) \otimes +2N \oslash +1\surd
\end{aligned} \tag{6.1}
$$

This analysis applies to the worst case when all spheres are kept until the end of the algorithm.

In comparison, the complexity of the geometric method described in Section 4.2 is as follows.

1. The calculation of $|\mathbf{r}_o\vec{\mathbf{x}}_c|^2$ needs *5* additions and *3* multiplications.

2. The dot product d_0 in Eq. 4.11 adds *2* additions and *3* multiplications;

3. To verify Eq. 4.10 and to check the sign of d_0, each requires a comparison, i,e, *2* comparisons.

4. In addition to the *1* addition and *1* multiplication involved in Eq. 4.12, *1* comparison is needed to check the relation between d_1^2 and r_s^2.

5. While Eq. 4.13 adds to the complexity *1* addition and *1* square root, $d_0 \pm d_t$ generate *1* additional addition.

In total, the number of operations for the geometric algorithm adds up to C_g.

$$
\begin{aligned}
C_g &= N(10 \oplus +7 \otimes +3 \oslash +1\sqrt{}) \\
&= 10N \oplus +7N \otimes +3N \oslash +1N\sqrt{} \qquad (6.2)
\end{aligned}
$$

6.2 A Comparison

If there is only one sphere in the scene, C_n and C_g are specialized by substituting $N = 1$ into Eq. 6.1 and Eq. 6.2 respectively.

$$
\begin{aligned}
C_n(1) &= 15 \oplus +23 \otimes +2 \oslash +1\sqrt{} \\
C_g(1) &= 10 \oplus +7 \otimes +3 \oslash +1\sqrt{} \qquad (6.3)
\end{aligned}
$$

$C_g(1)$ is better than $C_n(1)$ in the special case since the new algorithm needs five more additions and sixteen more multiplications than the geometric algorithm.

If there is more than one sphere in the scene, e.g., N spheres, although it is difficult to give an accurate comparison as it depends on both the scene and the machine on which the program runs, an estimation is still derivable from the general knowledge. In general, the cost of operations is in such an order:

$$\oslash \leq \oplus \leq \otimes \ll \surd \tag{6.4}$$

If, as a reasonable assumption[2], let $\oslash = \oplus = \otimes$, and $\surd = 10\otimes$, Eq. 6.1 and Eq. 6.2 have the following relation.

$$C_n(N) = 29 + 21N \tag{6.5}$$

$$C_g(N) = 30N \tag{6.6}$$

When $N > 3$, $C_n(N)$ is smaller than $C_g(N)$. This means that the new algorithm is better if a ray intersects more than three spheres in the scene. As a ray-traced image of moderate complexity involves a huge number of rays, the new algorithm is typically much faster than the geometric algorithm for such an image. Even if the cost of $\surd$ operation is only $3\otimes$, the new algorithm is superior, for a large number of intersections.

[2]It has been predicted that in the near future, $\oplus = \otimes$; and it was indicated by Haines[18] that $\surd$ takes *15-30* times more than $\otimes$.

Chapter 7

Implementation

In order to test the effectiveness of the new algorithm and collect data for comparison, the algorithm is implemented within an existing ray-tracing package. The implementation is actually based upon the public domain ray tracing package *Rayshade 4.0.6 (version 2)* developed by Craig E. Kolb and his colleagues at Princeton University from 1988 to 1992. The following sections give a brief overview of this package.

7.1 Understanding the Functionalities of *Rayshade 4.0.6*

The *Rayshade 4.0.6* package provides many functions or features which are explained in detail in the eight-chapter document entitled 'Rayshade User's Guide and Reference

Manual'[1].

There are actually two ways to provide input date into *Rayshade 4.0.6*.

1. Writing an input file according to a certain format.

2. Giving some simple options.

The second method is only employed for simple applications. For more complicated scene descriptions with many parameters, the first method is preferable since small changes in the scene can be made easily.

The functionalities or features provided in *Rayshade 4.0.6* are briefly indicated in the following list:

- Statistics Reporting
- Anti-aliasing
- Camera Position
- Field of View
- Depth of Field
- Stereo Rendering
- Light Source Types

[1]This document can be found under 'Doc/' directory within the *Rayshade 4.0.6* distribution.

- Shadows
- Primitives
- Aggregate Objects
- Constructive Solid Geometry
- Surface Description
- Atmospheric Effects
- Medium
- Transformations
- Texture Mapping
- Animation

A good way to understand a large software system initially is to get its 'structure graph' and analyze it in order to have a whole picture of the software skeleton in mind. Although a full structure graph is not available for this system, an old version 'structure graph' for 'Rayshade 1.0', is provided.

This 'structure graph' shows us that 'Rayshade' package can be divided into two main parts:

- Preprocessing

- Rendering

The 'Rendering algorithm' described in the 'structure graph' for 'Rayshade 1.0' is an older method and totally different from that in *Rayshade 4.0.6* package which uses the current popular 'raytracing algorithm'.

7.2 Understanding the source code of *Rayshade 4.0.6*

'Rayshade 4.0.6' is a program for creating raytraced images. It reads in a description of the scene to be rendered and produces a color image corresponding to this description. The description is actually an input file. In Rayshade, this file is read by the 'C-preprocessor' program provided by almost all UNIX systems, which compiles the input scene data using the 'yacc' program and puts the data into appropriate data structures for the 'Rayshade 4.0.6' program. The 'raytracing' algorithm will then take this data and begin the real 'raytracing' process. Finally, the raytraced image plane's pixel color value will be written into an output file. This output process employs the 'RLE' image representation and calls a set of functions developed at Utah university.

Any program can be abstracted into two parts, a *data structure* component and an *algorithm* component. The detailed explanation of 'Rayshade 4.0.6' program will discuss the two parts clearly.

7.2.1 Data Structure

MatrixMult(): matrix multiplication function.

MatrixInvert(): matrix inverse function.

VecTransform(): vector transformation function,
the transformations are restricted to translate,
rotate and scale.

MatrixCreate(): create a RSMatrix data structure

ListIntersect(): make a individual list to do intersection
testing with current ray.

ListBounds(): get a list's bounding box

GeomBounds(): get the object's bounding box

After the preprocessing stage, the scene description data is read into 'Rayshade 4.0.6' data structure as Figure 7.1.

7.2.2 Algorithm

The major relationships among the functions is as follows:

- main(argc,argv) (main.c)

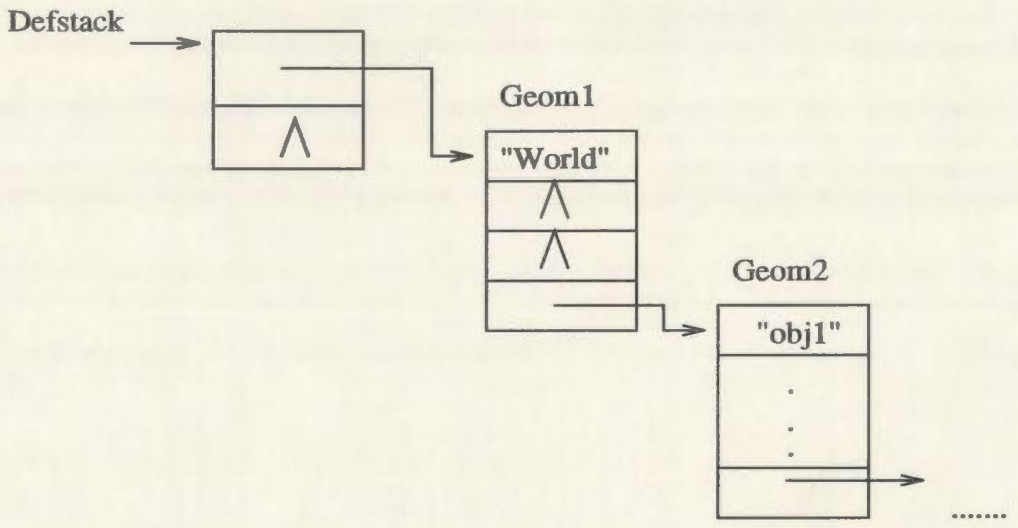


Figure 7.1: The object stack data structure after preprocessing stage of 'Rayshade 4.0.6'

- RSInitialize(argc,argv) (setup.c)

This is the preprocessing stage of 'Rayshade 4.0.6' program mentioned before and the detailed subroutine calling procedure could be checked out from the 'structure graph'.

- raytrace(argc,argv) (main.c)

  This is the ray trace processing function.

- RaytraceInit()

  The function finished some raytracing preparation, the most important part of these is to arrange the samples according to the user's requirement for anti-aliasing.

- TraceRay(ray, &hitlist, Epsilon, &dist)

This is the ray tracing algorithm. 'ray' here is the current ray vector. '&hitlist' is the result from raytracing processing containing all the objects which are hit by the current ray. 'Epsilon' is a variable for optimization purposes, it actually contains the smallest 't' value in '&hitlist' as return value. '&dist' is the distance from ray's origin to the intersection point.

- Intersect(World, ray, hitlist, mindist, maxdist)

  This is the function for conducting ray/object intersection testing. 'World' is the pointer to the data structure containing all the objects within the scene generated at the preprocessing stage. The other four variables are the same as in 'TraceRay' and will return back to 'TraceRay'.

- BoundsIntersect(ray, bounds, mindist, &nmaxdist)

  This is the function which has been modified in C-language and recompiled with the other functions for this implementation. This function tests 'ray' with specific 'bounds' from 'intersect' and determines if a hit might happen or not. That is the pre-determination part of the ray/object intersection algorithm.

- AddToHitList()

  This function adds all the hit objects into 'hitlist' for later processing.

- ShadeRay()

This function finds the exact intersection point from 'hitlist' and calculates the intersection point color value according to the illumination equations.

- Picture()

  This function puts the evaluated pixel color into appropriate the position in the output file.

7.3 Experimental Results

The new algorithm has been successfully implemented and tested on SUN^{TM} workstations. When the new algorithm was programmed in *Rayshade 4.0.6* , the algebraic and geometric algorithms were also implemented. To verify the theoretical analysis of the complexity and comparison, the time consumed by the program, namely 'user time', on the machine from the ray-volume intersection is recorded. Since *Rayshade 4.0.6* uses boxes as the bounding volumes, four types of data are obtained.

The experiment has two phases. In the first phase, images are rendered with a gradually increasing number of special spheres, which are randomly placed in the scene. The size of each image is 256×256, and the radius of each sphere is 0.167. By setting all spheres to non-reflecting and non-transparent objects, the number of spheres reflects the complexity of the scene in the direction of a random ray since every ray is an original ray. In the second phase, all four algorithms are tested with three 'big' scenes,

scene	user time of the algorithm *(seconds)*			
(sphere/tree)	*box-volume*	*algebraic*	*geometric*	*new algorithm*
3	7.72	6.32	6.85	6.45
4	10.04	8.31	8.94	8.19
5	12.02	9.95	10.80	9.71
6	15.13	13.72	13.27	12.42
7	18.63	15.93	16.32	14.60
8	24.15	19.41	20.51	19.46
9	26.96	22.26	23.27	20.20
10	29.31	25.12	25.28	23.75
11	32.52	27.08	26.60	25.19
12	37.61	31.71	33.04	30.43
13	42.64	35.29	36.82	34.76
14	46.40	37.49	39.22	36.70
15	47.60	40.27	40.76	39.55
16	52.18	43.53	44.42	41.61
17	53.20	46.08	46.61	44.22
18	55.91	48.97	47.83	45.39
19	59.79	51.14	52.04	47.86
buckyball	380.00	330.92	310.57	275.18
branch	1,649.84	1,390.46	1,375.02	1,104.82
tree	5,358.12	4,550.37	4,451.86	3,605.10

Table 7.1: The Results of Experiment

a *buckyball*, a *branch* and a *tree*. The object files of the *buckyball* and *tree* scene were down loaded together with the *Rayshade* package. The object file of the *branch* was modified from the *tree* file as part of the original tree. The *buckyball* is a 768×768 image with *150* primitives. The *tree* is a 983×768 image with $6,268$ primitives. The *branch* is also a 983×768 image but with only *1824* primitives.

In Table 7.1, the experimental results on a *SPARCstation 10-30* are combined

for both phases. The times are given in seconds. A graph of the performance is provided by Figure 7.3, where C_n, C_g, C_a, and C_b are the complexity in user's time for the new, geometric, algebraic, and box-volume algorithms respectively. The rendered images *buckyball*, *branch* and *tree* are shown in half size and grey levels in Figure 7.4, Figure 7.5, and Figure 7.6. The experiment shows that the new algorithm is not only theoretically but also in practice the best among the four algorithms in all tested cases.

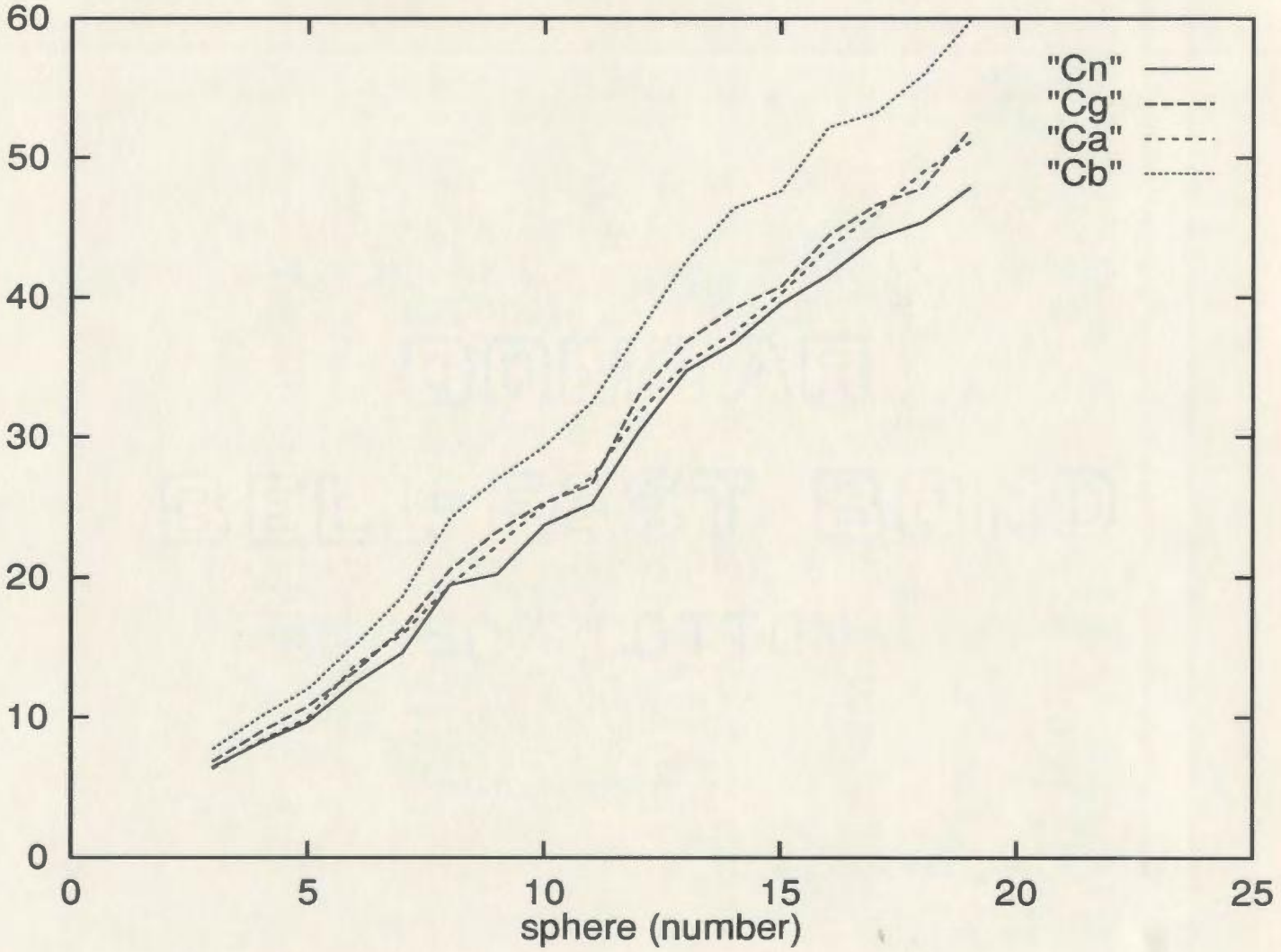


Figure 7.2: The Performance of the Algorithms

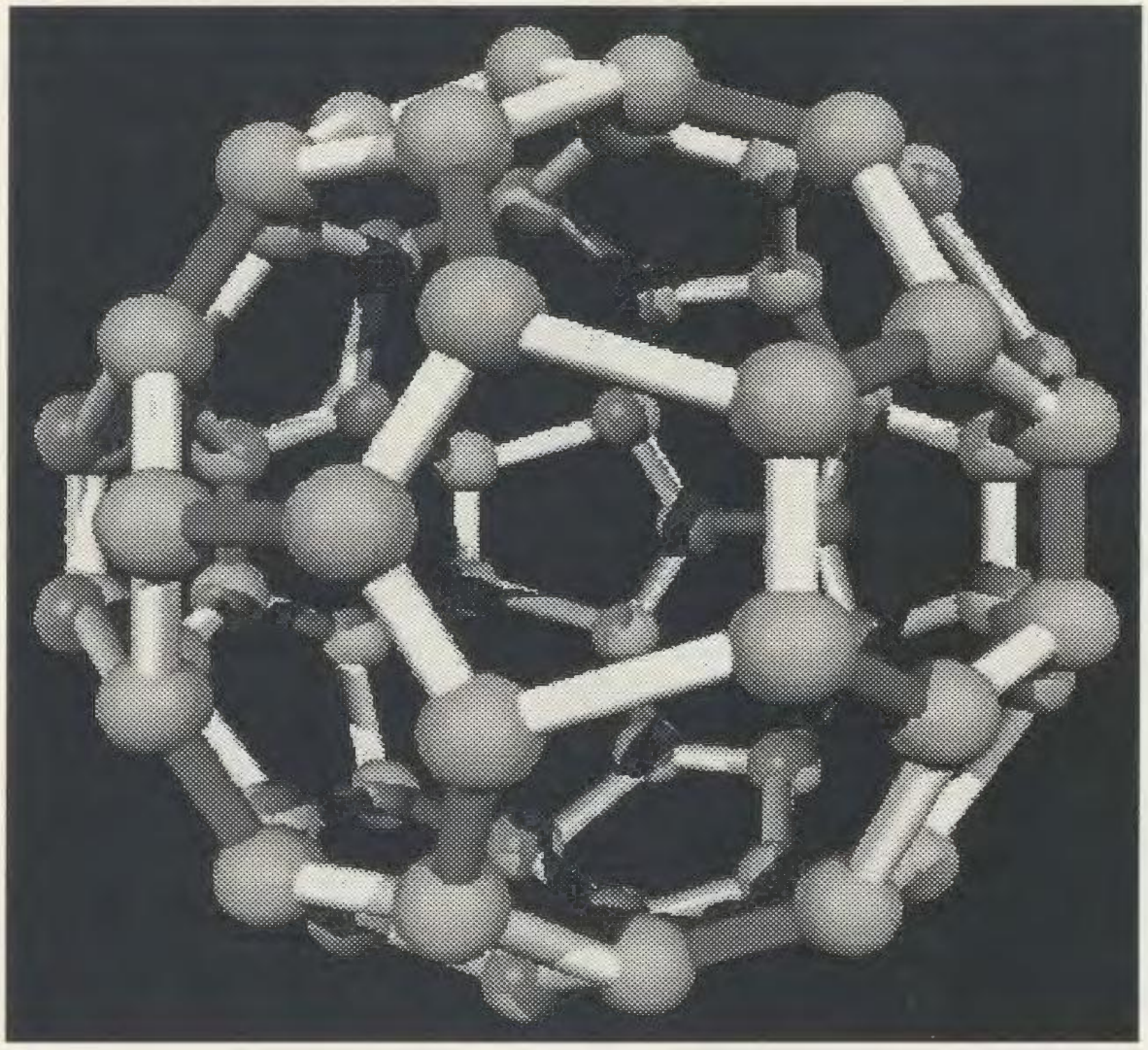


Figure 7.3: *Buckyball*, a Rendered Image

Figure 7.4: *Branch*, a Rendered Image

Figure 7.5: *Tree*, a Rendered Image

Chapter 8

Conclusion

In this thesis, the ray-tracing technique, a rendering algorithm for producing 'super-real' 2D images is discussed. A ray tracer generates a color value for each pixel by tracing the ray of light backward from the eye position to the sources in a 3D environment and simulates the light intensity at the pixel due to three physical processes, namely, local illumination, reflection and transparency. Ray tracing is widely applied in current graphics packages.

In a ray-tracing algorithm, the major time consuming processing is the calculation of the intersection point of the current ray with objects within a scene. Therefore, acceleration techniques for the ray/object intersection task were surveyed, concentrating particularly on bounding volume techniques. After an overview of bounding volume technology, the sphere emerged as the most useful volume because the balanced burden

for using it is the least, i.e. it has the simplest intersection calculation and is the easiest volume to create no matter what kind of object it bounds.

Two current ray/sphere intersection algorithms were discussed in detail. They are the algebraic and geometric solutions. A careful analysis of their algorithms leads to the observation that their *pre-determination* stage can be realized in another way. Thus, a new fast ray/sphere intersection algorithm has been discovered.

The new method uses a homogeneous transformation to simplify three-dimensional ray-sphere intersection into two-dimensional point-circle enclosure check, and thus speeds up the rendering of ray traced images. After comparing the time complexity of the new algorithm with the current most efficient geometric solution, it was shown that the more objects existing within a scene, the better the new algorithm would appear.

This theoretical analysis was also tested by implementing the ray/box algorithm and the ray/sphere algorithms with the algebraic, geometric, and the new solutions. The experimental results also show that the new algorithm is the best among them. Thus, it can be concluded with confidence that the new algorithm can be very useful in graphics applications relating to ray tracing rendering techniques.

Remaining work includes investigating the possibility of extending this algorithm to "bundled" ray tracing techniques; e.g., beam tracing[35] or cone tracing[36], and to animation systems[37].

Appendix A.

The C-Code of the New Algorithm

This appendix gives the c-code of the new algorithm developed in the thesis. It defines the function that checks if a ray hits a bounding sphere. The *Rayshade* package is available at "`http://www-graphics.stanford.edu/~cek/rayshade/rayshade.html`".

```
int BoundsIntersect(ray, bounds)
Ray *ray;
float bounds[2][3];
{
    static float r[4][4] = {
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0 };
    Vector sc, snc;                                  /* 3-D vectors */
    float s2r, sr;
    float l, m, n, x0, y0, z0;
    float tmp;
```

```
sc.x = bounds[0][0];                    /* initialize the sphere center */
sc.y = bounds[0][1];
sc.z = bounds[0][2];
s2r = bounds[1][0];                     /* squred sphere radius */
sr = bounds[1][1];                      /* sphere radius */
l = ray→dir.x;                          /* ray's direction */
m = ray→dir.y;
n = ray→dir.z;
x0 = ray→pos.x;                         /* ray's origine */
y0 = ray→pos.y;
z0 = ray→pos.z;
/* obtain the transformation matrix */
r[0][2] = l;
r[1][2] = m;
r[2][2] = n;
r[2][0] = -sqrt( 1 - n*n );
tmp = 1.0 / r[2][0];
r[0][1] = m * tmp;
r[1][1] = -l * tmp;
r[0][0] = n * r[1][1];
r[1][0] = -n * r[0][1];
```

```
    r[3][0] = -x0*r[0][0] - y0*r[1][0] - z0*r[2][0];
    r[3][1] = -x0*r[0][1] - y0*r[1][1];
    r[3][2] = -x0*l - y0*m -z0*n;
    /* calculate the 'z' component of the new sphere center */
    snc.z = sc.x*r[0][2] + sc.y*r[1][2] + sc.z*r[2][2] + r[3][2];
    if( snc.z+sr ≤ 0 )
        return FALSE;                          /* sphere is behind the ray */
    /* calculate the other two components of the new sphere center */
    snc.x = sc.x*r[0][0] + sc.y*r[1][0] + sc.z*r[2][0] + r[3][0];
    snc.y = sc.x*r[0][1] + sc.y*r[1][1] + r[3][1];
    if( snc.x*snc.x + snc.y*snc.y ≥ s2r )
        return FALSE;                          /* ray is out of the circle */
    else
        return TRUE;                           /* ray hits the sphere */
}
```

Bibliography

[1] Clark, J.H., Hierarchical geometric models for visible surface algorithms. *Commun. ACM* **19**(10), 547-554, October 1976.

[2] Fuchs, H., On visible surface generation by a priori tree structures. *Comput. Graph.* **14**(3), 124-133, July 1980.

[3] Fujimoto, A., Tanaka, T. and Iwata, K., ARTS: Accelerated Ray-Tracing System. *IEEE Comput. Graph. Appl.* **6**(4), 16-26, April 1986.

[4] Gervautz, M., Three improvements of the ray tracing algorithm for CSG trees. *Comput. Graph.* **10**(4), 333-339, 1986.

[5] Glassner, A. S., Space subdivision for fast ray tracing. *IEEE Comput. Graph. Appl.* **4**(10), 15-22, October 1984.

[6] Goldsmith, J. and Salmon, J., Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.* **7**(5), 14-20, May 1987.

[7] Jansen, F.W., Data structures for ray tracing. In *Data Structures for Raster Graphics, Proceedings Workshop* (eds L.R.A. Kessener, F.J. Peters, M.L.P. Lierop) pp. 57-73, Eurographics Seminars, Springer Verlag, 1986.

[8] Kaplan, M.R., Space tracing a constant time ray tracer. State of the Art in Image Synthesis (Siggraph '85 Course Notes), Vol. 11, July 1985.

[9] Kay, T.L., and Kajiya, J., Ray tracing complex scenes. *Comput. Graph.* **20**(4), 269-278, August 1986.

[10] Roth, S.D., Ray casting for modeling solids. *Comput. Graph. Image Process.* **18** 109-144, 1982.

[11] Rubin, S. and Whitted, T., A three-dimensional representation for fast rendering of complex scenes. *Comput. Graph.* **14**(3), 110-116, July 1980.

[12] Sweeney, M.A.J. and Bartels, R.H., Ray tracing free-form B-spline surfaces. *IEEE Comput. Graph. Appl.* **6**(2), 41-49, February 1986.

[13] Toth,D.L.,On ray tracing parametric surfaces. *Comput. Graph.* **19**(3), 171-179 July 1985.

[14] van de Hulst, H.C., *Light Scattering by Small Particles*, Dover Publications, New York, 1981.

[15] van Wijk, J.J., Ray tracing objects defined by sweeping planar cubic splines. *ACM Trans. Graph.* **3**, 223-237, (3), July 1984.

[16] Weghorst, H., Hooper, G. and Greenberg, D., Improved computational methods for ray tracing. *ACM Trans. Graph.* **3**(1), 52-69, January 1984.

[17] Yau, Mann-May and Srihari, S.N., A hierarchical data structure for multidimensional digital images. *Commun. ACM* **26**(7), 504-515, July 1983.

[18] Eric Haines, *Essential Ray Tracing Algorithms*, "An Introduction to Ray Tracing", Academic Press, 1989

[19] Alan Watt, Mark Watt, "Advanced Animation and Rendering Techniques,Theory and Practice", Addison-Wesley, 1993

[20] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, " Computer Graphics: Principles and Practices, second edition", Addison-Wesley, 1990

[21] Whitted, T., An Improved Illumination Model for Shaded Display, *Comm. ACM*, 26(6), 342-9. 1980

[22] Bui-Tuong Phong, Illumination for computer generated pictures. *Commun. ACM* 18(6), June 1975

[23] Kajiya, J.T., Ray tracing parametric patches. Siggraph '82

[24] R.A. Earnshaw and Norman Wiseman. *An Introductory Guide to Scientific Visualization*. Springer-Verlag, 1992.

[25] K.W. Brodlie, L.A Carpenter, and etc., editors. *Scientific Visualization: Techniques and Applications*. Springer-Verlag, 1992.

[26] M.W. Firebaugh. *Computer Graphics: Tools for Visualization*.W.C. Brown, 1993.

[27] J. Foley and et al. *Computer Graphics: Principles and Practice*. Addison Wesley, second edition, 1992.

[28] D. Hearn and M. P. Baker. *Computer Graphics*. Prentice Hall, second edition, 1994.

[29] W. Gellert and et. al. *The VNR Concise Encydopedia of Mathematics*. Van Nostrand Reomeold, second edition, 1989.

[30] J. Roe. *Elementary Geometry*. Oxford University Press Inc., New York, 1993.

[31] R.A. Earnshaw and D. Watson, editors. *Animation and Scientific Visualization: Tools and Applications*. Academic Press, 1993.

[32] Alan Wat and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, 1993.

[33] S.G. Hoggar. *Mathematics for Computer Graphics*. Cambridge University Press, 1992.

[34] X. Yuan. A mechanism of automatic 3D object modeling. *IEEE Trans. on PAMI*, 17(3):307-311, March 1995.

[35] P. S. Heckbert and P. Hanrahan. Beam Tracing Polygonal Objects, *Computer Graphics*, 18(3):119-127, 1984.

[36] J. Amanatides. Ray Tracing with Cones. *Computer Graphics*, 18(3):129-135, 1984.

[37] S. J. Adelson and L.F. Hodges, Generating Exact Ray-Traced Animation Frames by Reproduction, *IEEE Computer Graphics and Applications*, 15(3):43-52, 1995.