Space Time Ray Tracing using Ray Classification

Matthew Quail

Submitted for the partial fulfilment
of the requirements of the Bachelor
of Science with Honours

November 1996

Department of Computing
School of Maths, Physics, Computing and Electronics
Macquarie University

Abstract

Throughout the history of computer graphics, a very active area of research has been the attempt to create convincing, life-like images. Ray tracing is a technique for generating photorealistic images, and can easily support a wide range of natural phenomenon. However, producing a ray traced image is very expensive; and producing a ray traced animation is often prohibitively expensive.

This thesis looks at two techniques for efficiently producing ray traced animations; Glassner's space time hierarchical bounding volume technique, and my space time extension of Arvo and Kirk's ray classification. The results show that my method is more efficient; however Glassner's technique has much better memory performance. Based on some of the ideas of my method, some ways of improving Glassner's technique are suggested.

Contents

Chapter 1

Introduction

Ray tracing is a powerful technique for *image synthesis*, that is, creating a 2D image of a 3D world. Ray tracing can also simulate a wide variety of real world effects, producing images that look like a real photograph, called *photo-realistic* images. However, ray tracing is much more computationally expensive than conventional techniques, and creating an animation using ray tracing is often impractical.

The aim of this thesis is to investigate two methods for the efficient ray tracing of animations. The first is Glassner's *space-time hybrid hierarchal bounding volumes/space subdivision* method [1]. The second method is my space-time extension of Arvo and Kirk's *ray classification* [2].

1.1 Salient points of ray tracing

The term 'ray tracing' does not refer to a specific algorithm, but to a group of algorithms [3]. The two efficiency methods being investigated belong to one class of these algorithms. However, it will be necessary to give an overview of the whole of the ray tracing field (and rendering in general) to set the context for the class of algorithms that I will look at.

A framework is needed for conceptualising and describing an animation in computer graphics; this framework follows along the lines of how a motion picture is captured in real life. The content of a scene is made up of the objects in that scene plus the behaviour of the "virtual camera" that records the scene (a detailed explanation of the camera model used is given in Section 2.4). The scene is rendered into a set of frames or *images*, where an image is a rectangular array of pixels. The image (or image plane) is equivalent to the film plate of a real camera. Rendering an animation is the process of determining the colour of

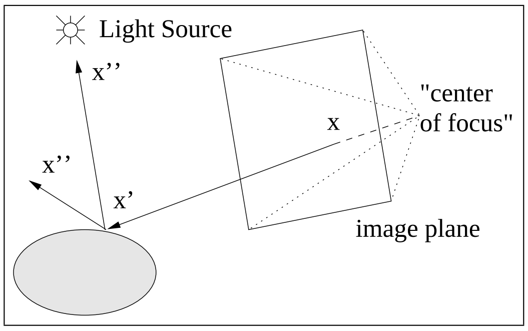


Figure 1.1: The colour of pixel at x is found by $I(x, x')$.

each pixel in each image.

An image in a camera is formed by the light that enters the camera and strikes the film. If the light in a scene can be modelled on a computer, then it is possible to create a synthetic image of a real scene. The light that hits a pixel on the image plane determines that pixel's colour[1]. This light can be determined using the *rendering equation*, developed by Kajiya [4] (see Figure 1.1),

$$I(x, x') = g(x, x') \left[\epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right]$$

where:

- $I(x, x')$ is the light that leaves point x' and arrives at point x.
- $g(x, x')$ is a 'geometry' term describing occlusion information between x and x'.
- $\epsilon(x, x')$ is the radiance of the emitted light from x' to x.
- $\rho(x, x', x'')$ is related to the light from points x'' that reflects from x' towards x.

Note that the rendering equation is recursively used to find the light reflected from the scene (points x'') to x via x'.

In practice, however, the rendering equation is extremely hard to solve. Ray tracing provides an approximation to the rendering equation. The approximation can be represented like this:

$$I(x, x') \approx \texttt{trace\_ray}(x, x')$$

$$\texttt{trace\_ray}(x, x') = g(x, x') \left[\epsilon(x, x') + \sum_{x'' \in \mathcal{I}} \rho(x, x', x'') \texttt{trace\_ray}(x', x'') \right]$$

[1]A concrete meaning to colour and light is not relevant to the aim of this thesis. I will simply talk about colour in an intuitive sense.

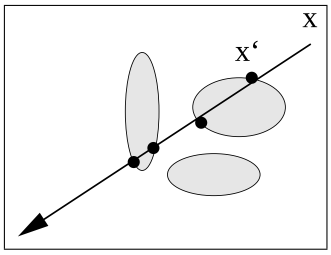


Figure 1.2: Intersection points of ray with scene.

where $\mathcal{I}$ is the set of "interesting" points, like light sources[2] and points that cause mirror like reflection.

To find the colour of a pixel in the image using ray tracing, a ray is traced from the pixel into the scene, *to find the intersection point x'*. Such a ray is called a *primary* ray. The evaluation of the rendering equation will recursively cause more rays to be traced. Each ray has an associated *generation*, which is its depth in the recursion. The application of the rendering equation is commonly called *shading.*

To summarise, a ray is traced into the scene to find the point x', which is *the closest point of intersection of the ray with the objects in the scene.* Once x' is found, the (approximated) rendering equation can be evaluated. Given all the points of intersection of a ray with the scene (Figure 1.2) , finding x' means searching for the closest point. In this way, the process of tracing a ray is a *spatial search for the closest intersection.*

1.2 Spatial search efficiency

A commonly reported figure is that 95% of rendering time is spent doing the spatial search for the closest intersection [5]. Improving this search will greatly improve the time taken to render the scene. The naive way to do this search is to test the ray against *every* object in the scene, then find the point that was the closest. This is called an *exhaustive* search, and has linear time complexity in the number of objects.

Arvo and Kirk broadly classified the acceleration techniques that can be applied to ray tracing [6]:

- *Faster intersections.* These techniques aim to reduce the average cost per ray. They can be further classified:
  - Try to obtain *faster ray-object intersections.*

[2]Rays that are traced to light sources are called *shadow rays.*

  - *Fewer ray-object intersections.*

- Create *fewer rays.*

- Attempt to group bundles of rays together as *generalised rays.*

By using 'generalised rays', the concept of a ray is replaced with a more general entity such as a cone or pencil. Although the essential concepts of ray tracing remain the same, it has the advantage of being able to consider many rays at once.

There are several ways to obtain 'faster intersections'. Efficient routines for computing the intersection of rays with objects is the most obvious, and can be used by any implementation. Complex objects could also be surrounded with a *bounding volume* that offers a faster intersection test than the object. In this way, many rays could be dismissed without having to intersect them with the more complex bounded object.

Each ray traced may spawn more rays when it is shaded. This forms what is called a *ray tree*. *Super-sampling* may be used to reduce aliasing in images; resulting in many rays being traced per pixel. Two ways for creating 'fewer rays' are adaptive ray tree depth control and statistical optimisations for super-sampling. Ray tree depth control ensures that the rendering equation is not infinitely recursed; two ways of controlling the depth are to stop when the tree gets to some hard limit, or to stop when the contribution of a ray to the colour of the primary ray falls below some threshold.

The category 'fewer ray-object intersections' consists of techniques that try to reduce the need for an exhaustive search for an intersection. Given a ray, these techniques return a subset of the objects in the scene, which contains (at least) all the objects that the ray might intersect.

Most ray tracing acceleration techniques fall into this category, including *bounding volume hierarchies*, *space subdivision* and *directional techniques*. The remainder of this section will give an overview of these three schemes.

1.2.1 Hierarchical bounding volumes

Using bounding volumes may decrease the cost of intersections, but it does not reduce their number; bounding volumes are essentially a 'faster intersections' optimisation. At best, bounding volumes will reduce the computation by a constant amount, but they do not improve the linear time complexity of an exhaustive search for an intersection. [6].

Rubin and Whitted [7] introduced the concept of nested hierarchies of bounding volumes, which theoretically improves the search to logarithmic complexity in the number of objects. By placing a *parent* bounding volume around a group of *child* bounding volumes, we can reject *all* the children objects from consideration if the ray doesn't intersect the parent bounding volume. Volumes can be arbitrarily nested, creating a hierarchy (or tree) of volumes, with a volume bounding the scene at the top and the objects themselves at the leaves.

It is now a question of *how* a hierarchy is built. It is not hard to see that some hierarchies will be better than others; one that does most of the pruning near the root of the tree will be able to dismiss much larger groups of objects. The hierarchy construction technique that will be discussed in this thesis is Glassner's hybrid adaptive space subdivision/bounding volume technique [1], which is presented in Chapter 2.

1.2.2 Spatial subdivision

Bounding volume hierarchies subdivide groups of objects, whereas spatial subdivision subdivides the space around the objects. The bounding box for the scene is divided up into smaller, axis-aligned boxes called *voxels*. Each voxel is associated with a *candidate list*: all those objects that are inside that voxel.

To intersect a ray with the scene, it is necessary to consider only those objects in the candidate lists of the voxels the ray passes through. Furthermore, if the voxels are visited in the order in which they are pierced by the ray, many intersection tests can be avoided once an intersection has been found [8].

There are two ways to subdivide space: uniformly and non-uniformly. Non-uniform subdivision has the advantage that it can increase the level of subdivision in regions with a high density of objects, but it is often a complex procedure to step from one voxel to another along the path of a ray. It is easy to step from one voxel to another when using uniform subdivision, but subdivision is independent of the layout of the scene, and is a disadvantage for scenes with non-uniform object distribution.

Spatial subdivision is a top down approach; the scene is divided from the top to group objects at the bottom. Hierarchical bounding volumes builds its structure from the bottom up; objects are grouped together progressively until the whole scene is bounded.

1.2.3 Directional techniques

Directional techniques attempt to take the direction of rays into account when constructing candidate sets. To see how this differs from other approaches, consider how ray direction is

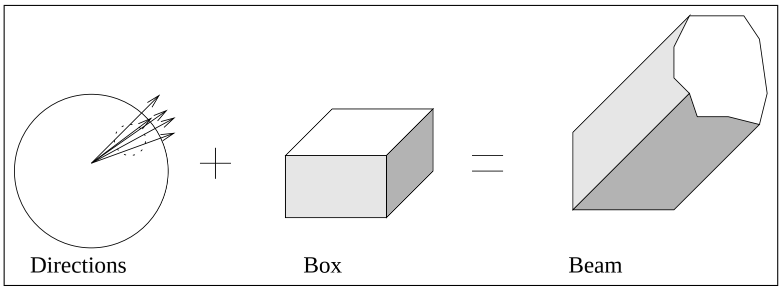


Figure 1.3: A beam in 3D.

used with spatial subdivision. Given each ray, the direction is used to find the set of voxels that the ray passes through. This process is done *for each ray.* Directional techniques take this consideration of direction out of the 'inner loop', and tries to do these calculations once.

The directional technique that is used here is Arvo and Kirk's *ray classification* [2]. A ray can be described as an origin and a direction. The origin of any ray will lie within the bounding box $B \subset \mathcal{R}^3$ of the scene. The direction can be described by a point on the unit sphere S^2. This means that rays can be paramaterised by elements of the 5-space $B \times S^2$.

As in 3D subdivision, this 5-space could be subdivided into small cells $C_i \in B \times S^2$. Given that C_i is small enough, these cells parameterise rays with similar origins and similar directions. Such a set of rays defines a *beam* in 3D, as in Figure 1.3 (adapted from [6]).

A candidate set of objects is associated with each cell C_i, which is precisely those objects that are inside that beam. To intersect a ray with the scene, the ray is *classified* as to what cell C_i it belongs, and then the objects in that cell are search for an intersection.

1.3 Summary

This chapter introduced those background concepts that are required for the remainder of the thesis. Chapter 2 will give a formal treatment of the theory that is required to implement both efficiency schemes. Chapter 3 will cover the design of the implementation, and Chapter 4 will detail the issues that arose from the implementation of the theory.

Chapter 2

Theory

The aim of this thesis is to compare two (animation) ray tracing efficiency schemes; Glassner's technique and my extension of Arvo and Kirk's. This chapter will describe the theoretical and technical details necessary to implement both schemes. An excellent introduction to the many other technical aspects of ray tracing is given by Glassner [9].

2.1 Tracing rays

A ray R is defined as all the points

$$R \equiv \begin{pmatrix} x & y & z \end{pmatrix}^T = \mathbf{o} + \mathbf{d}u, u > 0 \tag{2.1}$$

where $\mathbf{o} = \begin{pmatrix} o_x & o_y & o_z \end{pmatrix}^T$ is the origin of the ray, and $\mathbf{d} = \begin{pmatrix} d_x & d_y & d_z \end{pmatrix}^T$ is the direction of the ray. The direction $\mathbf{d}$ is unit length for primary rays, but we will see how this may not be true for rays in general.

The method for intersecting a ray with a surface depends on how that object is represented. The surface normal vector at the intersection point must also be obtained, for shading purposes. An example of sphere/ray intersection will be given here. Methods of intersecting rays with other surfaces can be found in [9].

The implicit equation for a sphere is

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0 \tag{2.2}$$

where $\mathbf{S_c} = \begin{pmatrix} x_0 & y_0 & z_0 \end{pmatrix}^T$ is the centre of the sphere, and r is the radius. Substituting x,y and z in Equation (2.1) into Equation (2.2), we obtain

$$(o_x + d_x u - x_0)^2 + (o_y + d_y u - y_0)^2 + (o_z + d_z u - z_0)^2 - r^2 = 0$$

which is a quadratic equation in u. This can be solved using the quadratic equation; yielding no real solution (ray missed sphere), one real solution (ray grazed sphere, which is ignored), or two solutions u_0 and u_1. Finding the smallest of these values greater than zero will give us the closest intersection of the ray with the sphere. Substituting this value of u into Equation (2.1) gives us the point $\mathbf{p}$ where the intersection occurs. The surface normal of the sphere at this point is given by the direction of $\mathbf{p} - \mathbf{S_c}$.

2.2 Modelling

A common practice in computer graphics is to design an object in some local co-ordinate system, and then place the object in the world co-ordinates of the scene with a *modelling transformation*. I will use the notation for transformations of Foley et al [10].

A transformation M is a 4×4 matrix representing a linear transformation,

$$M = \begin{pmatrix} & \mathcal{A} & & T \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{2.3}$$

where $\mathcal{A}$ is a 3×3 affine matrix, and T is a 3×1 matrix representing a translation. Transforming a point $\mathbf{x}_L$[1] in local co-ordinates by M results in the point in world co-ordinates

$$\mathbf{x}_W = M \cdot \mathbf{x}_L.$$

To transform a point from world to local co-ordinates, M^{-1} is used instead. Transforming a normal $\mathbf{a}_L$ to world co-ordinates by M gives [11]

$$\mathbf{a}_W = (M^{-1})^T \cdot \mathbf{a}_L.$$

Because of the form of M, a transformation will never change the homogeneous co-ordinate of $\mathbf{x}_L$ or $\mathbf{a}_L$.

Transforming a point $\mathbf{x}_L$ by M_1 then by M_2 is given by the concatenation of the two matrices:

$$\begin{aligned} \mathbf{x}_W &= M_2 \cdot (M_1 \cdot \mathbf{x}_L) \\ &= (M_2 M_1) \cdot \mathbf{x}_L. \end{aligned}$$

[1]The point $\begin{pmatrix} x & y & z \end{pmatrix}^T$ can also be specified as the homogeneous point $\begin{pmatrix} x & y & z & 1 \end{pmatrix}^T$. Notation will be abused slightly, and we will assume that either form can be converted to the other implicitly.

The inverse of the concatenation of matrices is

$$(M_n \cdots M_2 M_1)^{-1} = M_1^{-1} M_2^{-1} \cdots M_n^{-1}.$$

A modelling transformation is normally created by the concatenation of *elementary* transformations. Elementary transformations are affine transformation (rotations and scales, for example) and translations. The inverse of an elementary transformation is simple to compute: the inverse of a rotation is an equal rotation in the opposite direction. So, as a modelling transformation is built, the inverse can be computed along side it using the above identity.

Consider some surface S that is made up of the set of points $S \equiv \{\mathbf{x}_L\}$. The surface S_W in world co-ordinates that results from transforming S by M is

$$S_W = M \cdot S \equiv \{\mathbf{x}_W\} = \{M \cdot \mathbf{x}_L\}$$

A common technique is to store each object in the scene database in local co-ordinates and associate with each object a modelling matrix that transforms it to world co-ordinates. To intersect a ray with a world-object, the ray R_W is transformed to local co-ordinates by M^{-1}, giving

$$R_L = M^{-1} \cdot R_W \equiv M^{-1} \cdot \mathbf{o} + M^{-1} \cdot \mathbf{d}u.$$

Note that this may result in a ray direction that is no longer unit length. The ray R_L is intersected with the local-object, resulting in a value u_0. This u_0 is valid both as the point along R_L that intersects the local-object, and the point along R_W that intersects the world-object. The local co-ordinate normal $\mathbf{n}_L$ can be transformed to world co-ordinates by $\mathbf{n}_W = (M^{-1})^T \cdot \mathbf{n}_L$.

Note that in transforming the ray to local co-ordinates, and in transforming the normal back to world co-ordinates, only M^{-1} has been used. Once M has been computed, it is only necessary to store M^{-1} for use in ray/object intersections. As noted above, M^{-1} can be computed while M is being built.

2.3 Animation

There are two sides to animation in computer graphics. One is the interface between the animator and the scene[2]. This interface offers a more abstract or human-oriented

[2]Here, a scene means the whole state of the world during the animation. The state of the animation at a single instant is a *frame*.

specification of animation. This is typified by inverse kinematics, where an animator could make a character perform the complex process of walking just by specifying two points and a speed. Watt and Watt provide a good reference for this side of animation [12].

The other side to animation is the actual rendering of the scene. In this stage, it is not important how the scene was modelled. Only the explicit movements of the objects are important. Ray tracing is a rendering process, so it is concerned only with this aspect of animation. In this section, I will present a method for specifying an animation in this concrete manner.

The animation of an object in the scene is specified by a *dynamic* (or *temporal*) transformation $M(t)$. This has the same form as a *static* transformation M, except that the elements of the matrix are functions of time, instead of real numbers. If the animation runs over the time interval $[T_0, T_1]$, the elements e of $M(t)$ are $e \in \mathcal{F} = \{f | f : [T_0, T_1] \rightarrow \mathcal{R}\}$. Here, e could specify *any* such function, including piecewise defined functions.

An extra co-ordinate is used to represent the time of a point. The 4-point[3] $\begin{pmatrix} x & y & z & t \end{pmatrix}^T$ represents the 3-point $\begin{pmatrix} x & y & z \end{pmatrix}^T$ at time t. Equation (2.3) becomes

$$M(t) = \begin{pmatrix} & & & 0 & \\ & \mathcal{A}(t) & & 0 & T(t) \\ & & & 0 & \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}. \tag{2.4}$$

By itself, the 3-point $\begin{pmatrix} x & y & z \end{pmatrix}^T$ represents a point at any time (or no time).

At any time $t_i \in [T_0, T_1]$, $M(t_i)$ evaluates to a real valued matrix. This matrix gives the world co-ordinates position of the dynamic object at t_i. Because of the form of $M(t)$, $M(t_i)$ does not change the time co-ordinate of a 4-point. This means that dynamic transformations change an objects position in 3-space, not in time.

In general, a scene is rendered into n evenly spaced frames. At time t_i in frame i, the rays cast have origin $\begin{pmatrix} o_x & o_y & o_z & t_i \end{pmatrix}^T$ and direction $\begin{pmatrix} d_x & d_y & d_z & 0 \end{pmatrix}^T$. This means that rays travel instantly through time. At extremely large or small scales, the time component of the direction could be set to the speed of light in the database, allowing us to handle relativistic effects [13].

To intersect a ray with a dynamic object, the inverse transformation matrix of the

[3]This represents the homogeneous point $\begin{pmatrix} x & y & z & t & 1 \end{pmatrix}^T$.

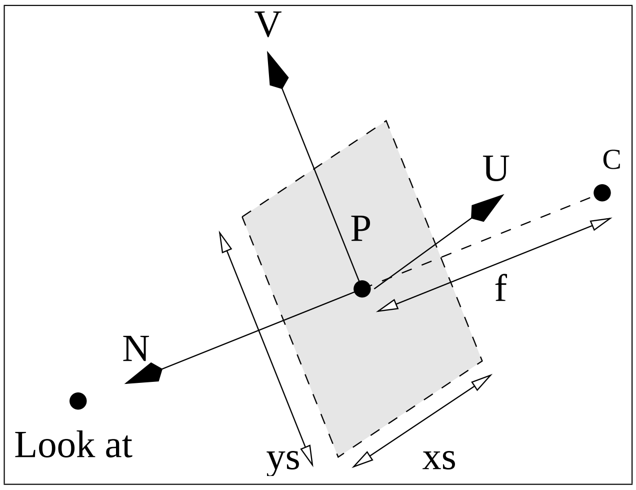


Figure 2.1: The camera model.

object is evaluated based on the time component of the ray, and this real-valued matrix is used to transform the ray to local co-ordinates, and the normal to world co-ordinates.

2.4 Camera model

The camera model used is adapted from the one suggested in Chapter 1 of Watt and Watt [12], see Figure 2.1. The animator specifies the camera position C, the 'focal' distance to the image plane f, a 'look at' position, an up direction V, and two scale values x_s and y_s that determine the size of the view window. From this information is determined the image plane normal N, and the 'right' direction U. Together, the normals N, V and U and the point P define the camera co-ordinate system.

The camera can be moved dynamically during a scene by associating a dynamic transformation with the camera. The normals N, V and U and the point P can be transformed at any point in time by $M(t)$, giving the camera orientation at that time. The constants f, x_s and y_s can also be specified dynamically as functions $f : [T_0, T_1] \rightarrow \mathcal{R}$.

At time t_i for frame i, the camera and all the objects could be transformed to their world positions at that time, and the frame could be rendered using traditional static ray tracing methods. However, the rendered animation will suffer from *temporal-aliasing.* Objects will jump from their position in one frame to their position in the next. *Motion blur* can be used to eliminate this effect.

Instead of setting each ray at time t_i, rays are initiated at some random time in the range $[t_i, t_{i+1})$. Several rays are cast per pixel, *stochastically sampling* across the time

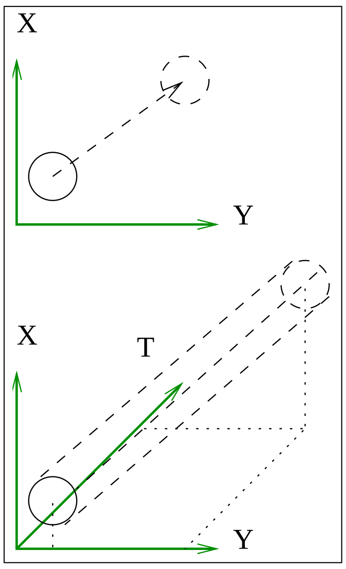


Figure 2.2: A moving 2D circle represents a sheared cylinder in 3D.

interval for that frame. This is an example of *distributed* ray tracing [14].

2.5 Space time

A moving 3D object can be considered as a static 4D object. To explain this, consider a 2D world, where a moving 2D object is a static 3D object. In Figure 2.2, a circle is to be translated along a line. If a time axis is added, then the path of the circle traces out a cylinder.

There is a benefit in considering animated objects as static space-time objects. If objects are considered in a purely 3D sense, then the only information that can be obtained from them is their position and orientation at a particular time. If they are represented as 4D objects, then that representation allows the object to be considered simultaneously over all of its life. A more fundamental explanation of the use of time as a fourth dimension is given by Taylor [13].

2.6 Bounding volumes

Both Glassner's method and Arvo and Kirk's ray classification require the use of the same type of bounding volumes, called *sets of slabs* [15]. The details of this type of volume is presented in this section.

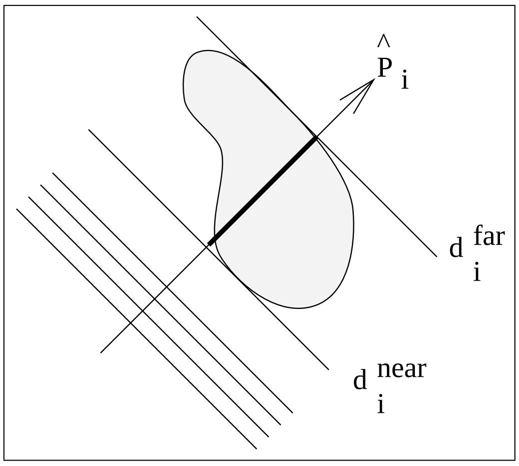


Figure 2.3: The set of planes defined by $\hat{P}_i$

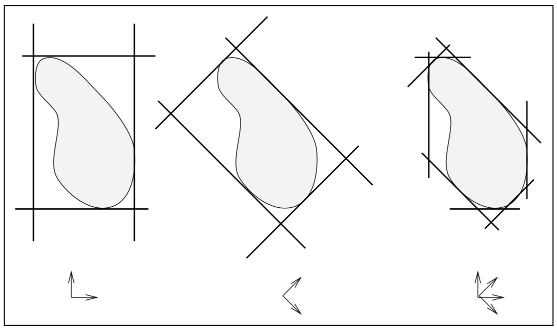


Figure 2.4: Sets of slabs with various normals.

An arbitrary plane in 3-space can be described by the implicit equation $Ax+By+Cz-d=0$. This describes a plane with normal vector $\hat{P}_i = \begin{pmatrix} A & B & C \end{pmatrix}^T$ lying d units from the origin. Fixing $\hat{P}_i$ and allowing d to vary describes all the planes normal to $\hat{P}_i$. From this set, we can choose two planes that bound an object, as in Figure 2.3. These planes can be described by the two values d_i^{near} and d_i^{far}. The region of space between these two planes is called a *slab*, and the normal vector that defines the orientation of the slab is the *plane-set normal*.

Consider a set of plane set normals $\{\hat{P}_1, \ldots, \hat{P}_m\}$, $\hat{P}_i \in \mathcal{R}^n$, that spans $\mathcal{R}^n$. Each normal $\hat{P}_i$ can be used to define a slab that bounds an object. The intersection of all such slabs will form a closed bound around the object.

This is shown in Figure 2.4 for $\mathcal{R}^2$. An object is bounded in three different ways; with the normals $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ and $\begin{pmatrix} 0 & 1 \end{pmatrix}^T$, with $\frac{\sqrt{2}}{2}\begin{pmatrix} 1 & 1 \end{pmatrix}^T$ and $\frac{\sqrt{2}}{2}\begin{pmatrix} 1 & -1 \end{pmatrix}^T$, and with all four normals.

A realistic scene may contain thousands or millions of objects, and storing the plane-set normals plus d_i^{near} and d_i^{far} for each object would require extraordinary amounts of memory. Instead, an arbitrary set of plane normals are chosen, and these normals are used for all objects. Only the pairs $(d_i^{\text{near}}, d_i^{\text{far}})$ are stored with each object. The trade off of preselecting plane-set normals means that we may not always create the optimal bounding volume for each object.

To construct a hierarchy of bounding volumes, it is necessary to find a parent bounding volume that bounds a set of child bounding volumes. With sets of slabs, the bounding volume of two child volumes is simply defined by the minimum of the d_i^{near} and maximum of the d_i^{far} values for each child.

In an animation, a moving 3D object represents a static 4D object[4]. It will be required that a 4D bounding volume be created for objects in a scene. Glassner suggests the following 12 plane set normals [1]: the principle planes

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

and the 'diagonal' planes

$$\frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}, \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ -1 \\ 1 \end{pmatrix}, \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \end{pmatrix},$$

$$\frac{1}{2}\begin{pmatrix} 1 \\ -1 \\ 1 \\ 1 \end{pmatrix}, \frac{1}{2}\begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}, \frac{1}{2}\begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \end{pmatrix}, \frac{1}{2}\begin{pmatrix} 1 \\ -1 \\ -1 \\ -1 \end{pmatrix}.$$

2.6.1 Computing bounding volumes

Consider the projection of an object onto the normal $\hat{P}_i$, shown as a dark line in Figure 2.3. The extents of that projection are exactly d_i^{near} and d_i^{far}. Computing the bounding volume of an object is simply the process of computing the extent of the projection of the object against each plane set normal (remember that the projection of $\mathbf{x}$ against the normal $\hat{P}_i$ is the dot product $\mathbf{x} \cdot \hat{P}_i^T$).

[4]A static 3D object also represents a static 4D object

Kay and Kajiay described methods of computing the bounding volume of several types of objects, where the objects had *static* modelling transformations associated with them. I will present a technique for computing the bounding volume for a general surface that has a *dynamic* transformation associated with it.

A surface S can be defined by the set of points $\{\mathbf{x}\}$ constrained by

$$\begin{gathered} g(\mathbf{x}) = 0, \\ t \in [T_0, T_1]. \end{gathered} \tag{2.5}$$

The 4D surface resulting from applying the modelling transformation $M(t)$ to S is the surface

$$S' \equiv M(t) \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix}$$

Using Equation (2.4), projecting S' onto $\hat{P}_i$ gives the function f,

$$\begin{aligned} f \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix} &= \hat{P}_i^T \cdot \left(M(t) \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix} \right) \\ &= \left(\begin{pmatrix} \hat{P}_x & \hat{P}_y & \hat{P}_z & \hat{P}_t & 1 \end{pmatrix} \cdot M(t) \right) \begin{pmatrix} x \\ y \\ z \\ t \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} A(t) & B(t) & C(t) & \hat{P}_t & \hat{P}_i^T \cdot T(t) \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ t \\ 1 \end{pmatrix} \\ &= A(t)x + B(t)y + C(t)z + D(t) \end{aligned} \tag{2.6}$$

Now, the function f constrained to the equations in (2.5) gives us the projection of S' onto $\hat{P}_i$. The extent of this projection is exactly the extrema of f constrained to

(2.5). This type of problem is known as a *constrained extrema* problem [16], and can be solved *analytically* using the method of Lagrange multipliers. However, the functions of t in Equation (2.6) are arbitrarily complex, and the constrained extrema cannot be found easily in practice.

The solution to this problem is to replace $M(t)$ with $M_L(t)$, where each element of $M_L(t)$ is a piecewise linear approximation of the corresponding element in $M(t)$. There is precedent for doing this linearisation [17]: Glassner did this in the implementation of his space-time article, and Pixar's Renderman assumes that motion is linear between frames.

Equation (2.6) is now only linear in t, and so can easily be solved using Lagrange multipliers. This is described in Chapter 4.

2.6.2 Ray-volume intersection test

The intersection of a ray with a slab yields an interval along the ray. The point along the ray $R = \mathbf{a}u + \mathbf{b}$ which intersects one of the planes in the slab is:

$$u = \frac{d_i - \hat{P}_i \cdot \mathbf{b}}{\hat{P}_i \cdot \mathbf{a}}. \tag{2.7}$$

This equation can be used to find both u^{near} and u^{far}. The endpoints may not be oriented correctly; if the denominator in Equation (2.7) is negative, then the roles of the near and far values must be reversed.

After computing the interval of intersection of the ray with each slab, we find the intersection of these intervals. This intersection is simply the maximum of the near values and the minimum of the far values. The ray misses the bounding volume if the intersection of the intervals is null; that is, if $u^{\text{near}}_{\text{max}} > u^{\text{far}}_{\text{min}}$.

There are two optimisations that can be performed on this process. One is that $\hat{P}_i \cdot \mathbf{b}$ and $\hat{P}_i \cdot \mathbf{a}$ in Equation (2.7) can be computed just once for each ray. The other involves detecting early on that the ray misses the volume before having to evaluate Equation (2.7) for all the slabs.

2.7 Glassner's space-time scheme

To efficiently ray trace animated scenes, Glassner proposed the space time ray tracing concept: dynamic 3D objects are rendered as static 4D objects [1]. Glassner also presented a hybrid hierarchical bounding volume/spatial subdivision technique as the acceleration scheme for his implementation. This technique will be discussed in this section.

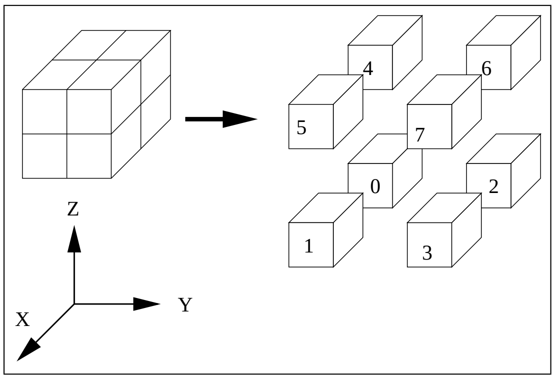


Figure 2.5: Subdivision and enumeration of a 3D box.

2.7.1 Building the hierarchy

The advantage of bounding volume techniques is that child volumes can be dismissed from consideration if a ray misses the parent volume. Only if a ray intersects the parent volume must child volumes be inspected. If volumes are allowed to overlap, then when inspecting the child nodes, it is not sufficient to simply look at the nearest node, because it may not contain the nearest object [15]. This indicates that it is hard to build 'good' hierarchies of bounding volumes.

Space subdivision is also an advantageous technique. The hierarchy created by adaptive subdivision is excellent: no cells at any given level overlap, and the subdivision is dense only where the database is dense. However, rectangular prisms offer poor bounding volumes compared to other types such sets of slabs.

The two techniques are complementary in the strengths and weaknesses: bounding volumes offer poor hierarchies but tight bounds, while adaptive subdivision offers good hierarchies but poor bounds. Glassner's technique uses the excellent hierarchy generated by space subdivision as a guide to build the tree of tight bounding volumes. The method for generating the hierarchy follows this rule: "space subdivision down, bounding volumes up".

A bounding box is found for the scene, and subdivision begins. A subdivision criterion (discussed below) is evaluated for the box and its contents. This determines if the box needs to be subdivided. Figure 2.5 shows how a 3D box is subdivided into 8 smaller cells. A 4D box is similarly subdivided into 16 cells.

The subdivision process is recursively repeated on each sub-box, until the scene has been adequately subdivided. For each cell, a bounding volume is created that contains the objects in that cell, *within the bounds of that cell.* This can be visualised by creating a

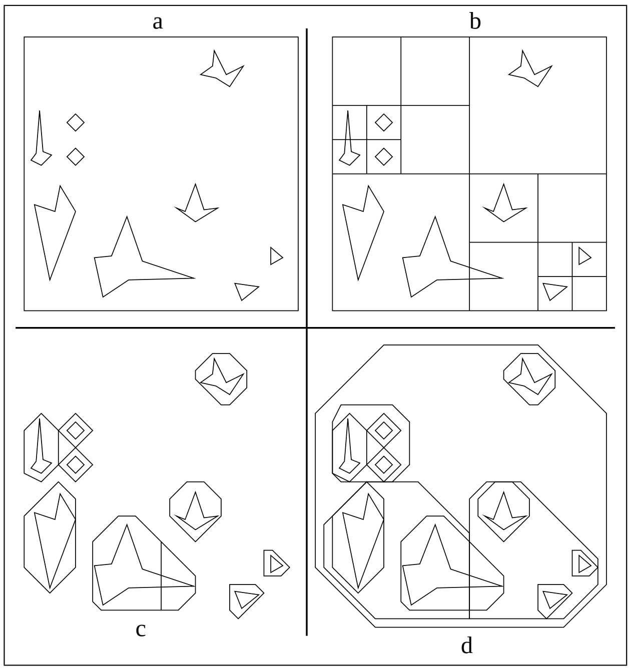


Figure 2.6: (a) a scene with 9 objects. (b) adaptive subdivision applied to the scene. (c) bounds for each object, within each cell. (d) final hierarchy built from bounds in (c) and tree structure in (b).

bounding volume for the objects in the cell, then clipping the bound to lie within the cell. See Figure 2.6 (taken from [1]).

As the subdivision recursion returns, a bound is built for each cell. This bound contains the bounds of all the child cells of that cell. This results in a tree of bounding volumes that has the non-overlapping hierarchy of spatial subdivision and the tight bounds of bounding volume techniques.

Glassner used equal subdivision [18] to subdivide the scene, and sets of slabs as the bounding volume. A hybrid subdivision technique was used; at upper levels, subdivision occurred if there were more than 3 objects in a cell. After that, the cell was still divided if the ratio of the volume enclosed by the objects to the volume of the cell was less the 0.3.

2.7.2 Ray/scene intersection

To find the intersection of a ray with the scene, the ray is tested first against the top bounding volume. If the ray misses this bound, then it misses all the objects in the scene. Otherwise, the ray is tested against the child volumes.

Because the volumes do not overlap, the ray can be tested against the first child volume along its path. If the ray does not hit any objects in that bound, or misses the bound, then the next volume along the path is checked. If the ray does not intersect any objects in the child volumes along the path of the ray, then the ray does not intersect any objects in the parent volume. If an intersection is found in a child volume, the point of intersection must be tested to see if it lies inside the child cell. If it does not, then that intersection must be discarded to avoid returning an intersection that is not the first along the ray.

2.7.3 Other details

One of the operations when subdividing the scene is to test if an object is inside a cell. However, ray tracing is only concerned with the surfaces of an object. Therefore, an object should be associated with a cell only if the *surface* if the object is in the cell [18].

An intersection must be rejected if it outside of the child cell that is being inspected. This may mean that a ray is tested against the same object several times, as the ray moves from cell to cell. To avoid computing the same intersection twice, a unique *tag* is associated with each ray. If a ray does intersect an object but is rejected because the point does not lie within the cell, then the tag and the intersection point are stored with the object. Before testing a ray against an object, the ray's tag is compared with the object's, to see if the test has already been performed.

2.8 Ray classification

This section will detail Arvo and Kirk's ray classification technique [2], and will be based on the ideas in Section 1.2.3.

The space of rays $B \times S^2$ is divided up into subsets $E_1, \ldots, E_m$. A set of candidate objects C_i is associated with each E_i. This candidate set is the complete set of objects that rays in E_i might intersect.

Spherical co-ordinates are difficult and inefficient to work with, so Arvo and Kirk specified the direction of a ray with the *direction cube*.

2.8.1 Direction Cube

Consider the axis-aligned cube of side two centred at the origin of any ray. Any distinct ray direction will correspond to a distinct point on the surface of the cube that is the intersection of the ray and the cube, as shown in Figure 2.7.

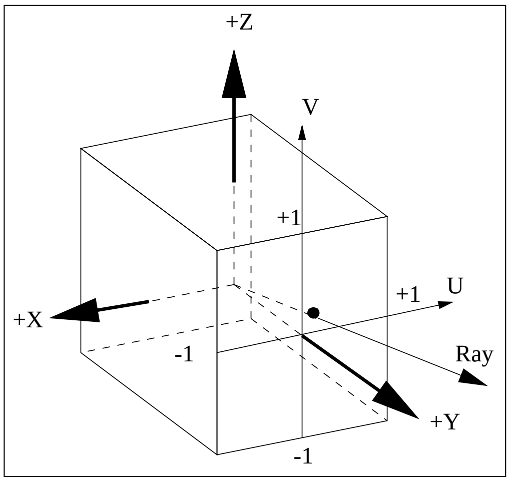


Figure 2.7: The direction cube.

The direction of a ray can now be specified by the 2D (u, v) point of intersection plus which dominant axis the ray intersected. This becomes the space

$$\mathcal{D} = [-1, 1] \times [-1, 1] \times \{+X, -X, +Y, -Y, +Z, -Z\}.$$

The 5-space of rays is now $B \times \mathcal{D}$. A procedure for converting a direction vector in $\mathcal{R}^3$ to a direction in $\mathcal{D}$ is given in [6].

A *hypercube* $E_i \in B \times \mathcal{D}$ now defines a beam in 3D which is an unbounded polyhedra with at most nine faces. In general, $E_i \in B \times S^2$ does not define a beam that is polyhedral. A polyhedral beam simplifies the process of classifying an object inside or outside a beam.

2.8.2 Arvo and Kirk's algorithm

The ray classification algorithm consists of five processes:

- computing B, the bounding box for the scene.
- *Hierarchy creation.* Selecting the subsets $E_1, \ldots, E_m$ which subdivide $B \times \mathcal{D}$ into disjoint volumes.
- *Candidate set creation.* Computing C_i for each E_i.
- *Ray classification.* Computing the hypercube E_i that a given ray lies in, and returning the candidate set for that hypercube.
- *Candidate set processing.* Given a ray and a candidate set, determining the closest ray intersection.

Hierarchy creation

Binary space subdivision is used to create the hypercube hierarchy, each axis of $B \times \mathcal{D}$ being divided exactly in half at each level. After enough subdivision has occurred, any ray can be classified to lie within a hypercube of arbitrarily small diameter.

To create an efficient hierarchy, subdivision should occur heavily in regions of $B \times \mathcal{D}$ that are dense in rays. However, the density characteristics of cast rays are determined as the scene is being rendered, so this information is not available in a pre-processing step. So instead, the hierarchy is created during rendering, using *lazy evaluation*.

Subdivision only occurs on demand, when the 5D co-ordinates of a ray are found to lie within a hypercube E_i that is too large. There are two heuristics that determine when subdivision terminates. One is that subdivision stops when the size of the candidate set falls below a given threshold; this achieves the goal of finding a small number of objects that a ray could possibly intersect. The second heuristic is to stop when the size of the hypercube falls below a given size (or the hypercube is at a certain depth in the hierarchy); this allows the cost of creating a candidate set to be amortized over may rays [2].

Candidate set creation

Given a hypercube, its candidate set is all the objects in the scene which intersect the set of rays which define the hypercube. An object should be classified in the candidate set of a hypercube if the object intersects the beam defined by the hypercube.

As an initial step, $B \times \mathcal{D}$ is subdivided along each of the six dominant axis. This gives six hypercubes $B \times [-1, 1] \times [-1, 1]$. All the objects in the scene are associated as an initial candidate set for each of these hypercubes.

For space efficiency, a candidate set is not created at every level in the hierarchy. When a hypercube is subdivided along an axis, the resulting beams usually overlap substantially, and are quite similar to the parent beam. Arvo and Kirk used the strategy of subdividing along each of the five axes before creating a new candidate set. So, up to $2^5 = 32$ hypercubes may inherit their candidate set from the same ancestor, but significant memory savings are gained. Also, due to lazy evaluation of the hierarchy, not all descendants are created.

Arvo and Kirk suggested three ways to classify whether an object was in a beam or not. Each method performed this check using the objects bounding volume, instead of the object itself. The method used in my implementation classifies the corners of the bound against the planes that define the beam. The other methods they suggested were to use linear programming, and to approximate the objects and beams with spheres and cones, respectively.

Ray classification

The dominant axis of a ray is the axis corresponding to the co-ordinate of the direction vector with the largest absolute magnitude. To find the hypercube E_i associated with a particular ray, the initial hypercube $B \times [-1, 1] \times [-1, 1]$ associated with the dominant axis of the ray is used as a starting point. That hypercube is the top of the hierarchy, and is traversed to find E_i. This yields the candidate set C_i. The process of traversing the hierarchy may also invoke the lazy evaluation of parts of it.

Arvo and Kirk noted that a particular generation of rays are likely to be classified to the same hypercube. This is because of *coherency*: for example, primary rays all have the same origin and similar direction. To exploit this coherency, the last hypercube referenced in each generation of rays is cached. New rays are first checked against the cache. Only if the ray is not associated with the hypercube in the cache is the hierarchy traversed. Although hierarchy traversal is very efficient, the cache check only requires ten comparisons.

Candidate set processing

Naively, a candidate set could be searched exhaustively for the closest intersection. Note that all rays in a beam share the same dominant axis. Objects in a candidate set can be sorted along their minimum extent along this axis. By processing the objects in ascending order, the tail of the list can be dismissed if an object is reached whose entire extent is beyond a known intersection.

The sorting of the objects can be done once for each dominant axis, during the initial subdivision. Since each child hypercube has the same dominant axis as its parent, it can inherit the correct order.

2.8.3 Five dimensional adaptive subdivision

One of the drawbacks with ray classification is that two parameters must be supplied with the scene to control the creation of the hierarchy. Simiakakis presented a method called *Five dimensional Adaptive Subdivision* (FAS) where these parameters are automatically computed by the ray tracer [8].

If a hierarchy is too deep, then too much time will be spent in hierarchy traversal. However, the candidate sets for a shallow hierarchy are very large, causing too much time to be spent in candidate set processing. This identifies two competing factors in hierarchy creation.

Simiakakis developed some expressions that tried to balance these factors. Statistics about the hierarchy were gathered as the program was running. Every 10 scan lines these statistics were used to adjust the parameter controlling the maximum tree depth. Simiakakis found that rendering time was relatively unaffected by the minimum candidate set size parameter; a value of 6 to 8 was optimal for most scenes.

Simiakakis also uncovered some other interesting results. The hypercube cache hit rate is very high, at about 60% to 90%. He also suggests that some of the results in Arvo and Kirk's article [2] cannot be correct unless some other technique was also used, such as a bounding volume hierarchy. Speer states that this "clouds the issue of data-structure culling efficiency" [19]. In [6], Arvo and Kirk mention that "it is also possible to use object hierarchies for the efficient creation of candidate lists".

The memory use of ray classification is very high. Simiakakis presents some methods for reducing this memory requirement, by using more memory efficient data structures, and by pruning not-recently-used branches of the tree when memory runs out.

2.9 Space time ray classification

Glassner extended traditional 3D hierarchical bounding volumes by adding an extra dimension that represented time. Ray classification can be extended in a similar and natural way. As well as having a origin and direction, a ray can also have a time. The space of rays now becomes $B \times [T_0, T_1] \times \mathcal{D}$, which is a 6 dimensional space.

Arvo and Kirk's algorithm works in space time with little change. Binary subdivision now occurs along six axes instead of five, and beams are tested against the 4D bounds of objects.

These are the only changes that need to be made to Arvo and Kirk's algorithm, but other changes can be made. It is possible to implement a 6D version of Simiakakis' FAS, to gain automatic tree depth control. However, the measurements that need to be taken to implement FAS are very complex, and such an implementation is beyond the scope of this project. Instead, a minimum candidate set size of 6 and a maximum tree depth of 5 was used, as suggested by Simiakakis.

2.9.1 Smart T division

Consider a beam E_i that needs to be subdivided. Normally, subdivision would occur along each axis. If the beam does not contain any moving objects, then the $2^5 = 32$ subdivisions on either side of the T axis will be exactly the same. To save on the number of subdivisions that occur, and hence the size of the hierarchy, the T axis is only subdivided if the candidate set C_i for E_i contains moving objects.

This means that parts of the scene that remain static will have a smaller hierarchy, and require less memory. There should also be more cache hits in that region; since the hierarchy is smaller there will be higher probability of consecutive rays lying in the same beam.

2.9.2 Pruning

FAS uses a memory saving scheme called pruning. If memory runs out, then a section of the hierarchy that hasn't been used lately is deleted. However, it is possible that the deleted part of the hierarchy will need to be recreated later, doubling up on some calculations. The pruning method I present here only prunes those parts of the hierarchy that are guaranteed never to be used again.

Once a frame has been rendered, the renderer continues on to render consecutive frames. If frame i, which is over the interval $[t_i, t_{i+1})$, has been completed, then no more rays that have a time component less than t_{i+1} will be traced. Therefore, it is safe to prune those parts of the tree that lie wholly within the interval $[T_0, t_{i+1})$. This pruning can happen at the end of each frame.

Since past parts of the hierarchy are deleted, and future parts have not been created yet due to lazy evaluation, then the total memory being used at any particular time will be approximately equal to the amount actually needed for that time interval. This means that the memory requirements for a scene will be independent of its duration.

Chapter 3

Design

This chapter gives a high level look at the design an implementation of the ray tracer built for this thesis.

There are two main reasons that an implementation is necessary to conduct the investigation that is the aim of this thesis. The most obvious is so that empirical results about the performance of both efficiency schemes can be gathered. However, building a ray tracer is also useful because it identifies the implementation issues that aren't obvious from the presented theory or literature. These issues are discussed in detail in Chapter 4.

3.1 Requirements

The requirements of the implementation are as follows:

- Produce ray traced animations.
- The program is primarily being used to test the spatial search algorithms. The algorithm that the program uses should be able to be changed or "plug-and-played" easily.
- Be able to gather a useful range of testing parameters (testing is discussed in Chapter 5).
- In view of obtaining some measure of photorealism, a range of object primitives and other important graphical effects should be supported.
- Provide an easy method for specifying the contents of a scene.

- An important requirement is that the implementation be efficient. This will influence the design, as well as decisions such as programming language.
- Portability; the program will be developed on a Sun Solaris system, but the testing will be performed on an SGI IRIX system.

3.2 Analysis of requirements

This section looks at the requirements in light of forming them into an implementation.

Firstly, complete photorealism must be compromised; it is beyond the scope of this project to create an industry quality photorealistic renderer. The advanced features that are supported are motion blur, anti-aliasing and the possibility of an arbitrary amount of primitive types[1].

Traditionally, modelling is supported by associating a modelling transformation M with an object (as described in Chapter 2). Animation can be supported by associating a dynamic transformation $M(t)$ with animated objects.

The program can be designed in such a way that it is modular, allowing different components to be changed or swapped with no change to the rest of the system. However, it is also important to be able to collect statistics, irrespective of what modules are currently "plugged" in.

3.3 Design

Ray tracing programs are commonly described using a functional model, but a very elegant object-oriented description is used for a particular subset of the system [20].

The main function of a ray tracing system is to evaluate the (approximation to the) rendering equation for each primary ray (Figure 3.1).

- The *Camera* module generates primary rays. These are passed to the *Kernel*, which returns the colour found by the rendering equation.
- The Kernel calls the *Object Manager* to find the closest intersection point x', given the ray R. The Kernel then calls the *Shader* to evaluate the rendering equation.

[1]Though, only spheres and polygons were implemented.

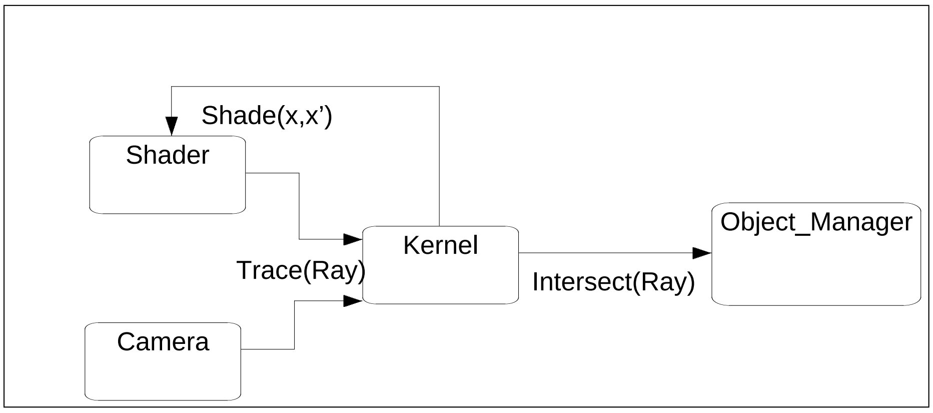


Figure 3.1: The modules that evaluate the rendering equation. Arrows indicate function calls.

- While computing evaluating the rendering equation, the Shader may need to recursively trace more rays.
- The role of the Object Manager is to store all the objects and to (hopefully efficiently) search for the closet intersection point of any given ray.

The Kernel's role is to manage the interaction of the other modules, and to collect statistics about the performance of the system.

Figure 3.1 describes the "inner loop" of a ray tracer. The system as a whole operates as in Figure 3.2:

- When the program is started, the Kernel calls the Scene Specification module, which defines the scene. This is a user written module that uses an API provided by the kernel for the adding of objects, light sources and a camera, plus any other relevant initialisation information.
- Once the scene has been specified, the list of objects are passed to the Object Manager, which performs any pre-processing that is required.
- The Kernel then repeatedly calls the Camera module to render each frame.

A polymorphic object oriented model is used to facilitate modelling, animation and an arbitrary number of primitive types (Figure 3.3). The "Object" class represents a primitive in the scene. It presents the following methods:

- `intersect(Ray)`: determines if the ray intersects this object. Returns the u value of the intersection if an intersection occurs.

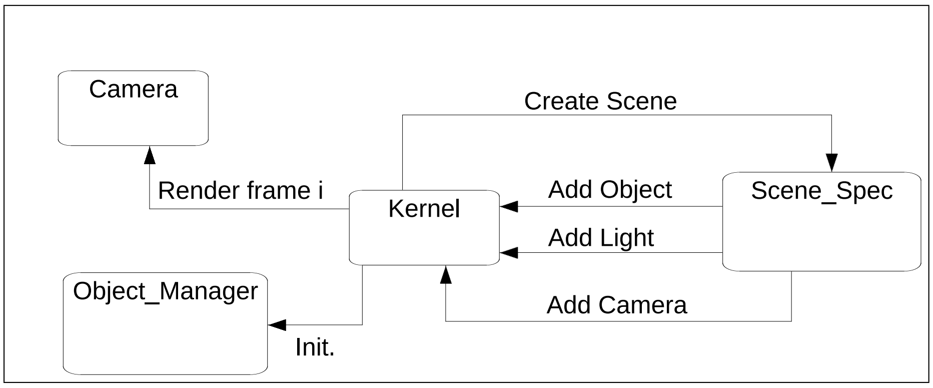


Figure 3.2: The top-level diagram.

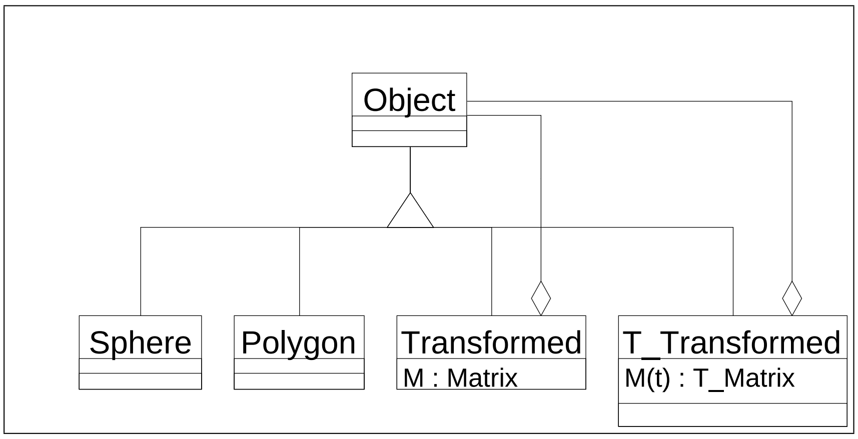


Figure 3.3: The "object" (primitive) object model.

- `aux_info(Intersect_Info)`: returns shading information (such as the surface normal) for a previously computed intersection with this object.
- `compute_bound()`: computes the bounding volume for this object.

The `intersect()` routine is used by the Object Manager to test rays against objects. Once the closest intersection has been found, then the Shader uses `aux_info()` to find the normal. This way, the normal is only computed for the intersection point that needs to be shaded.

An abituary number of primitives can be implemented by creating the appropriate subclasses from Object. Static and dynamic modelling is supported through subclasses of Object. The methods for these classes convert world co-ordinates to local before passing the request on to its child object.

3.4 Implementation

The program was written in `C++` in a Solaris environment at Macquarie University, and recompiled and run (for testing purposes) under IRIX on the SGI Power Challenge at the NSW Centre for Parallel Computing [2]. `C++` was used as the programming language because it offers object oriented features as well as ease of code hand-optimising.

Each time the program is run, the user may want to use different Object Manager and Scene Specification modules. To automate this process, a script file was created that compiles the user supplied Scene Specification source file, and links this with the desired Object Manager and the rest of the program, to produce an executable. The program is then run, and the testing output is recorded and saved.

The Whitted shader [21] was used to approximate the rendering equation. To shade a point, this technique traces one *shadow* ray for each light source, one *reflection* ray, and one *transparent* ray (if the object is transparent).

All the source code for the ray tracer is given in Appendix A.

[2]At the Parallel computing and Visualisation Laboratory

Chapter 4

Implementation Issues

Chapter 2 offers a fairly comprehensive review of the literature relevant to this thesis. Most of the ray tracing literature offers the theoretical basis for the method/algorithm being presented; then gives the results of an implementation of the method, without much treatment of the implementation itself.

This lack of implementation detail may be because the theory presented itself is implementation oriented, and implementation is a simple extension of the presented material. However, I found that there were important, non-trivial implementation issues that were never mentioned in the literature. This chapter outlines these issues, and other issues that arise with the methods I have presented, and details my solutions to them.

4.1 Linearisation of motion

Computing the 4D bounding volume of an animated object involves finding the constrained global extrema of $f\left(\begin{pmatrix} x & y & z & t \end{pmatrix}^T\right)$ in Equation (2.6). However, the functions of t in this equation may be arbitrarily complex in general, and no closed form solution to find the extrema exists.

The functions of t arise from the product $\hat{P}_i \cdot M(t)$, where $\hat{P}_i$ has real valued entries. If it is known that the entries of $M(t)$ are linear in t, then $\hat{P}_i \cdot M(t)$ will also be linear in t, as will f. Similarly, if the elements of $M(t)$ are *piecewise* linear functions of t, then f will be piecewise linear in t too. For example,

$$f(\mathbf{x}) = \begin{cases} f_1(\mathbf{x}) & T_0 \leq t < t_1 \\ \vdots & \\ & \\ f_n(\mathbf{x}) & t_n \leq t \leq T_1 \end{cases}$$

The extrema of f are then simply

$$\max(f) = \max_{i=1\ldots n} \{\max(f_i)\}$$

and

$$\min(f) = \min_{i=1\ldots n} \{\min(f_i)\},$$

and it will be shown how these extrema can be computed for spheres.

To get f in this form, $M(t)$ is replaced by $M_L(t)$ for each object, where the elements of $M_L(t)$ are piecewise approximations of the elements in $M(t)$. To linearise each element of $M(t)$, the interval $[T_0, T_1]$ is uniformly divided into n smaller intervals. Any value of n could be used, higher values of n corresponds to a closer approximation. To find n, I used the expression $n = \#\text{frames} \times$ `seg_factor`, where `seg_factor` is supplied by the user. This puts the quality of the linear approximation in the hand of the user. A value of `seg_factor` between 4 and 8 sufficed for scenes with very complex movement. A value of 1 means that motion is linear between frames.

4.1.1 Computing the extrema of a sphere

.

To compute the bounds of a sphere, g in Equation (2.5) becomes

$$g(\mathbf{x}) = (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0. \tag{4.1}$$

The method of Lagrange multipliers [16] tells us that the extrema will occur at $\mathbf{x}$ when $\nabla f(\mathbf{x}) = \lambda \nabla g(\mathbf{x})$. Remembering that $A(t)$, $B(t)$, $C(t)$ and $D(t)$[1] in Equation (2.5) are linear in t, we get

$$\begin{pmatrix} A(t) \\ B(t) \\ C(t) \\ a_0x + b_0y + c_0z + d_0 \end{pmatrix} = \lambda \begin{pmatrix} 2x - 2x_0 \\ 2y - 2y_0 \\ 2z - 2z_0 \\ 0 \end{pmatrix} \tag{4.2}$$

[1]These have the form $A(t) = a_0t + a_1$.

which gives

$$\begin{array}{rcl} x - x_0 & = & \frac{A(t)}{2\lambda} \\ y - y_0 & = & \frac{B(t)}{2\lambda} \\ z - z_0 & = & \frac{C(t)}{2\lambda} \end{array} . \tag{4.3}$$

Substituting (4.3) into (4.1) gives

$$\begin{aligned} \frac{A(t)^2}{4\lambda^2} + \frac{B(t)^2}{4\lambda^2} + \frac{B(t)^2}{4\lambda^2} - r^2 &= 0 \\ A(t)^2 + B(t)^2 + C(t)^2 &= 4\lambda^2 r^2 \\ \lambda &= \pm\frac{1}{2r}\sqrt{A(t)^2 + B(t)^2 + C(t)^2}. \end{aligned} \tag{4.4}$$

Substituting (4.3) into the last row in (4.2) gives

$$\begin{aligned} a_0\left(\frac{A(t)}{2\lambda} + x_0\right) + b_0\left(\frac{B(t)}{2\lambda} + y_0\right) + c_0\left(\frac{C(t)}{2\lambda} + z_0\right) + d_0 &= 0 \\ \frac{1}{2\lambda}(a_oA(t) + b_0B(t) + c_0C(t)) + a_0x + b_0y + c_0z + d_0 &= 0 \end{aligned}$$

and since $a_0x + b_0y + c_0z + d_0 = 0$ from (4.2),

$$\frac{1}{2\lambda}(a_oA(t) + b_0B(t) + c_0C(t)) = 0$$

and $\lambda \neq 0$ from (4.3), the extrema for f occur when

$$a_oA(t) + b_0B(t) + c_0C(t) = 0$$

or, solving for t, when

$$t = -\frac{a_0a_1 + b_0b_1 + c_0c_1}{(a_0^2 + b_0^2 + c_0)^2}. \tag{4.5}$$

Given t, both values of λ can be found by (4.4), yielding two pairs $\begin{pmatrix} x_a, y_a, z_a, t \end{pmatrix}^T$ and $\begin{pmatrix} x_b, y_b, z_b, t \end{pmatrix}^T$ from (4.3). Substituting these pairs into f gives the extrema of f.

In (4.5), it could be the case that $a_0 = b_0 = c_0 = 0$. But this means that $A(t)$, $B(t)$ and $C(t)$ are constants. Therefore, from (4.2), $d_0 = 0$, and (4.4) gives $\lambda = \pm\frac{1}{2r}\sqrt{a_1^2 + b_1^2 + c_1^2}$. From this we can find pairs from (4.3) and extrema for f.

4.2 Box/surface testing in Glassner's method

As described in Section 2.7, Glassner builds a nested hierarchy of 4D bounding volumes to speed up the spatial search. 4D spatial subdivision is used to guild the building of

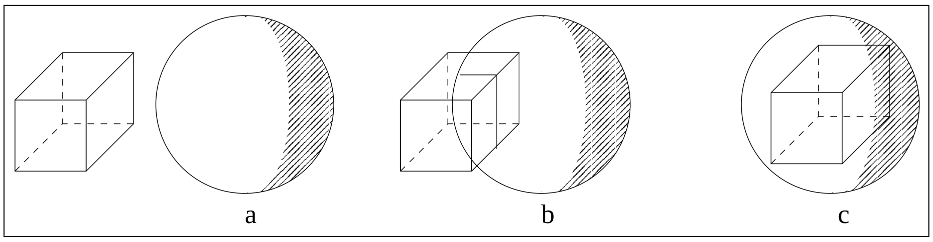


Figure 4.1: Classification of a sphere and a cell. None of the surface is within the cell in (a) or (c), only (b) classifies the surface within the cell.

the hierarchy. A fundamental operation that needs to be performed when doing spatial subdivision is to test whether the surface[2] of an object is inside a particular cell. This is shown in Figure 4.1.

Consider an object that has an associated modelling transformation M (this is not even a dynamic transformation). To classify the surface of the object against the cell, the cell could be transformed to local co-ordinates. Given the from of M, an axis aligned rectangular cell in world co-ordinates will be transformed to a parallelepiped in local co-ordinates.

To classify a polygon against such a parallelepiped, a linear transformation L is found that transforms the parallelepiped to a the unit cube centred at the origin, and transforms the polygon to some other polygon. L can be found by considering the transformation as a change of basis. Voorhies [22] presents a method that classifies the surface of a triangle against the unit cube centred on the origin. A polygon can trivially decomposed into triangles, as in Figure 4.2. If any of the triangles are in the cell, then the the polygon is in the cell. This method is sound in theory, however, my implementation did not consistently classify polygons correctly. I believe that this is because of floating point truncation and round-off errors that build up in the application of the modelling transformation and then the change of basis.

Arvo [23] presents a method that classifies the surface of an *axis-aligned* ellipsoid against an axis-aligned cell. A sphere could be classified against a parallelepiped by transforming the parallelepiped to an axis-aligned cell, but the sphere could be transformed to a *non*-axis-aligned ellipsoid, which Arvo's method does not handle. I was unable to come up with a solution to this problem, so Arvo's method is used as is, and wrongly classifies some spheres. There are two points of interest here; one is that this seems a serious problem, but

[2]Rays only intersect surfaces, not volumes, so a ray may only intersect an object within a cell if the surface of that object is within the cell.

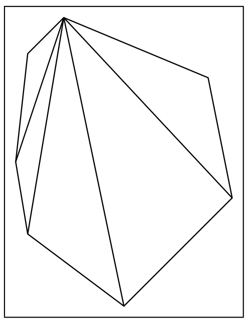


Figure 4.2: A polygon decomposed into triangles.

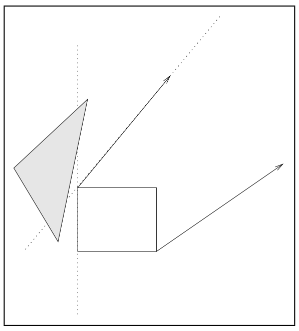


Figure 4.3: Object/beam classification (2D example). An object is classified as inside the beam since it has points inside all half-spaces defining the beam, even though the object is not in the beam itself.

no reference is made to it in *any* of the subdivision literature that I read. The other point is that this is a hard problem, and *dynamic* transformations haven't even been considered yet!

4.3 Object/beam classification

To classify an object against a beam in ray classification, the corners of the object's bounding volume are tested against the planes that define the beam. This is a simple test, but not rigorous; it may classify a bound as in the beam when it is in two of the planes, but not in the actual beam, as in Figure 4.3. Theoretically this classification is not erroneous; it never classifies an object outside the beam when it is actually inside. However, my implementation does occasionally missclassify objects against beams. I believe this to be a combination of machine truncation errors and some buggy code.

4.4 Comments

Linearising the description of movement is an elegant solution to the problem of creating 4D bounds for moving objects. The linearised movement will only be noticeable with objects that move in a very nonlinear way within one frame, for which the approximation level can be adjusted accordingly. Also, objects that move greatly within one frame exhibit extreme motion blur, so that it is hard to see what that object is doing anyway; in this way, linearisation does not significantly stand in the way of the quest for photorealism.

One way solve the problem of surface/cell classification is to relax the strength of the test. Instead of erroneously classifying a surface outside when it is actually in, a different algorithm might be developed that sometimes classifies an object in when it is out, but always classifies it as in when it *is* in. Still, there is no indication of such an algorithm in the literature, but such an algorithm would probably be easier to develop than the exact solution.

Chapter 5

Empirical investigation

The aim of this project is to investigate two spatial search methods for ray tracing. Since the aim of these methods is to improve on the linear time complexity of an exhaustive search, it is important to look at the efficiency of the methods as part of the investigation.

5.1 Testing

What to test

As with a search on an array, the number of comparisons performed is the measure of complexity for a spatial search. In ray tracing, the comparison is a ray/object intersection test. The number of ray/object intersection tests can be used to indicate how much improvement over an exhaustive search has been achieved, and which method offers the bigger improvement.

Another indication of efficiency is the time it takes to render an animation. A particular search scheme may offer much better than linear time complexity, but pays for this with a terribly expensive pre-processing step. Ideally, run time would be the best indication of efficiency; however, the run time is highly dependent upon how well the program has been optimised. Although every attempt was made while programming to ensure an efficient implementation, a working, finished program was more important than an perfectly efficient but unfinished one.

I have used the following areas of interest in deciding on what to test for.

- *Time efficiency.* The number of ray/object intersections shows how good the search scheme is, but balancing this, the rendering time indicates the true cost of the scheme.

- *Memory efficiency.* Memory use is an important issue in both Glassner's method and Arvo and Kirk's ray classification.

- *Algorithm performance.* It is useful to know the details of all the tasks the program is performing. For example, the number of frames in the animation, the number of objects, or the proportion of primary/reflection/transmission and shadow rays. It is also useful to know the details of the search algorithms, like the cache hit rate in my method, or the depth of the hierarchy in Glassner's.

How to test

To collect the results of the testing, the program must be used to render some scenes. The types of scenes that are used for testing are important; the test scenes should reflect qualities that are common to the types of scenes that are rendered by real photo-realistic renderers, and the scenes should not be tailored to suit the advantages of any of the particular search methods.

Eric Haines [24] proposed standard set of *static* scenes, called the Standard Procedural Database (SPD), that could be used to test the performance of renderers. This set of scenes attempted to encapsulate a range of image attributes, including the types of primitives used, the shading complexity (number of shadows, reflections and transparent objects) and the placement of objects in the scene.

However, a set of scenes are needed that encapsulates the range of *animated* scenes commonly rendered. Haines and I [25] determined the following attributes for the possible types of animated scenes:

- A moving camera through a fixed scene; for example, an architectural work through.

- Localised object movement; a particle spinning or the hands of a clock moving.

- Wide ranging movements; a spaceship/car moving around the scene

- A mixture of the above.

Haines [25] also noted that the SPD does not have fringe objects; objects that are not a main part of the scene, but just sit at the edges.

The following animations were used as the test scenes:

- *Night City.* The camera swoops down along a street of a "virtual" city. This is mainly a "moving camera, fixed scene" animation; however, it also has some fringe objects, representing flowing traffic.

- *Mountains.* This is an adaptation from the SPD. Crystal balls move around in front of a fractal mountain. This is a fixed scene, with some small localised movement. This scene has a large transparent component.

- *Boldly.* The starship Enterprise races around the scene, with two very complex geometric fringe objects, taken from the SPD. This is an example of wide ranging movement with complex fringe objects.

- *Party-cools.* Ten coloured spheres move around the scene randomly. This scene has complex, wide ranging movement, with no fringe objects.

It should be noted that the version of "Boldly" that was used as a test scene accidently differs slightly from the one I designed; the Enterprise moves too fast, and appears as just a blur. I could not change it to more realistic motion since I discovered discrepancy too late. However, this scene is still as useful test scene, it is now an extreme example of fast, wide ranging movement.

5.2 Results

Table 5.1 shows the statistics for the various scenes. All scenes were rendered once for each search method, at 12.5 frames per second, with motion blur and anti-aliasing using 3×3 jittered super-sampling [26]. The frames were then colour quantised to a 256 colour palette and stored as an animation. See Appendix A on how to obtain these animations.

The results of the test scenes when run using an exhaustive search are given in Table 5.2. Even though an exhaustive search for the closest intersection point was used, each object was surrounded with a bounding volume; the ray was first tested against the bound, and then only tested against the object if the ray intersected the bound. This same bounding test was also used with the other search methods. The "Ray/BV tests" figure indicates the number of ray/object comparisons; "Ray/Object tests" indicates how many of these comparisons actually required a test with the underlying object, and not just the bound.

Table 5.3 shows the results for Glassner's method. For the "Night City" animation, the program failed to finish, as it was stopped after running for 5 hours and not having finished the pre-processing step. The time taken for the Object Manager to initialise is also given; this is the time taken to build the space-time hierarchical bounding volumes.

The results for my space time ray classification method are given in Table 5.4. The Object Manager initialisation time was always less than 1 second for my method. The performance of the classification cache is also given. Table 5.5 gives a comparison between the two methods. The figures reported are the ratio between my method and Glassner's; values less than one indicate that mine was faster.

Sample frames from the animations are given in Figures 5.1 to 5.4. Memory usage is given in Table 5.6. Actual memory usage will depend on the efficiency of the structures used, and no attempt was made to optimise the size of the structures. Therefore, memory usage is reported as the number of nodes used.

Animation	#Rays $\times 10^3$	P/R/T/S $\times 10^3$	#Frames	#Objects
Night City	169477	45000/28729/0/95748	125	401
Mountains	26773	8640/2517/5481/10134	24	516
Boldly	183587	135000/15451/0/33136	375	354
Party-cool	27890	27000/487/0/403	75	10

Table 5.1: Statistics about the test scenes (the number of rays is broken down into the number of primary, reflection, transparent and shadow rays).

Animation	Night City	Mountains	Boldly	Party-cool
Ray/BV tests $\times 10^6$	67960	13815	2208	278
Ray/Object tests $\times 10^6$	453	106	518	83
Ray/Object hits $\times 10^6$	249	246	447	49
Rendering time (sec)	19803	4180	23952	815

Table 5.2: Results using exhaustive search.

Animation	Night City	Mountains	Boldly	Party-cool
Ray/BV tests $\times 10^6$	–	433	1687	408
Ray/Object tests $\times 10^6$	–	93	293	158
Rendering time (sec)	–	1324	6677	1469
OM init. time (sec)	–	30	403	28

Table 5.3: Results using Glassner's method.

Animation	Night City	Mountains	Boldly	Party-cool
Ray/BV tests $\times 10^6$	1420	167	431	153
Ray/Object tests $\times 10^6$	452	64	284	83
Rendering time (sec)	4269	656	4539	1119
Cache hit rate	71%	33%	88%	30%

Table 5.4: Results using my method.

Animation	Night City	Mountains	Boldly	Party-cool
Ray/BV tests	–	0.38	0.25	0.37
Ray/Object tests	–	0.68	0.96	0.52
Total Rendering time	–	0.48	0.64	0.74

Table 5.5: Performance ratio of my method to Glassner's.

Animation	Night City	Mountains	Boldly	Party-cool
Glassner's	–	299	185	2
Mine	779	542	1323	1529

Table 5.6: Memory required by the hierarchies. Figures represent number of nodes ×1000.

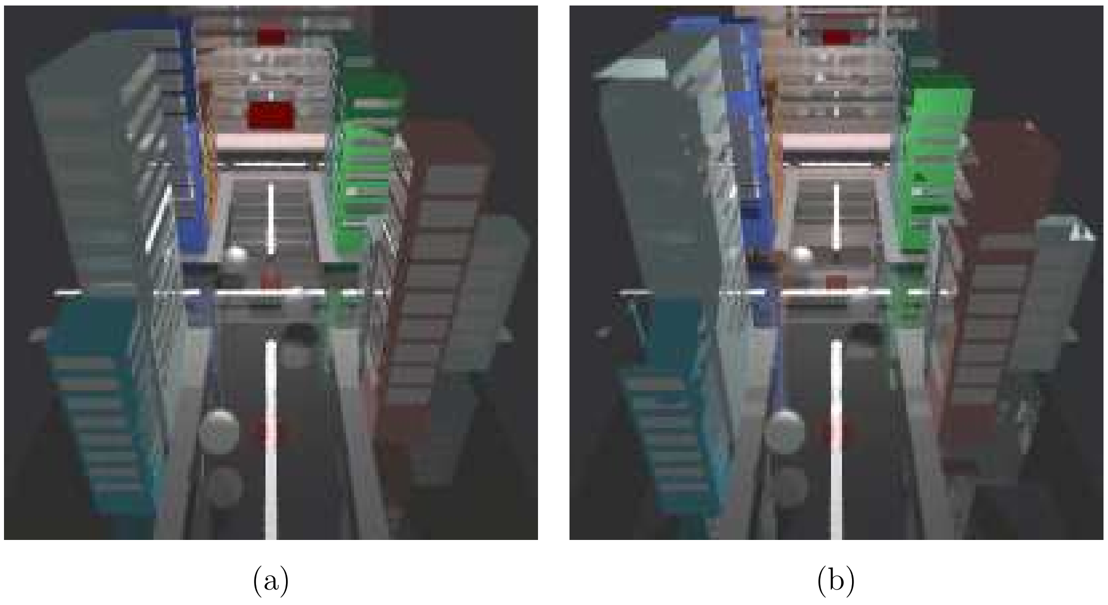


Figure 5.1: The same two frames from "Night City", using (a) exhaustive and (b) my method. Notice that some of the polygons are missing in (b).

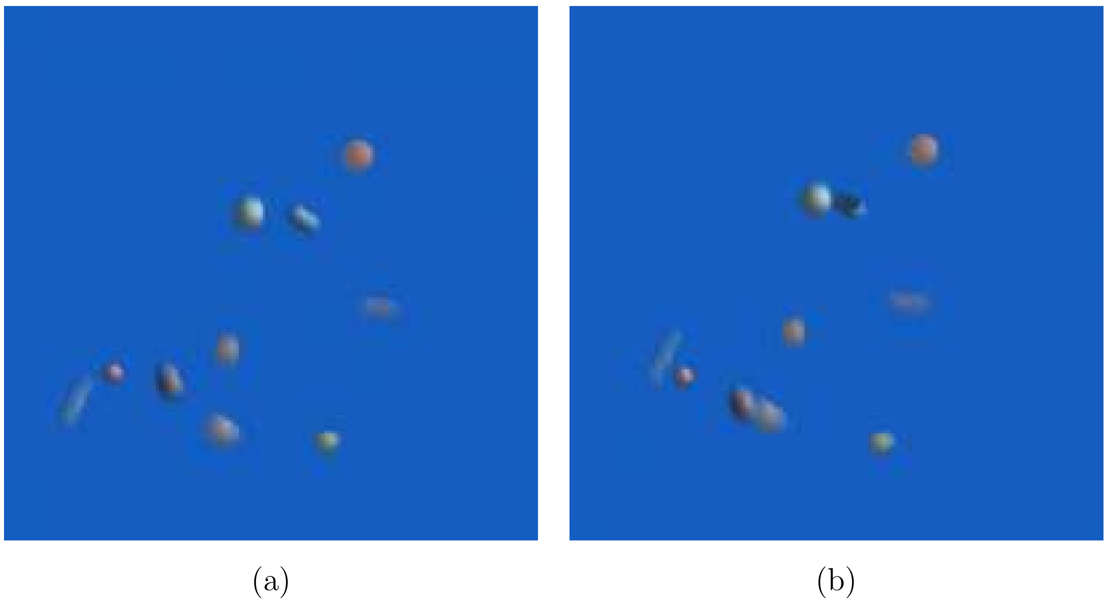


Figure 5.2: Two consecutive frames from "Party-cool". Note the motion blur on some of the balls.

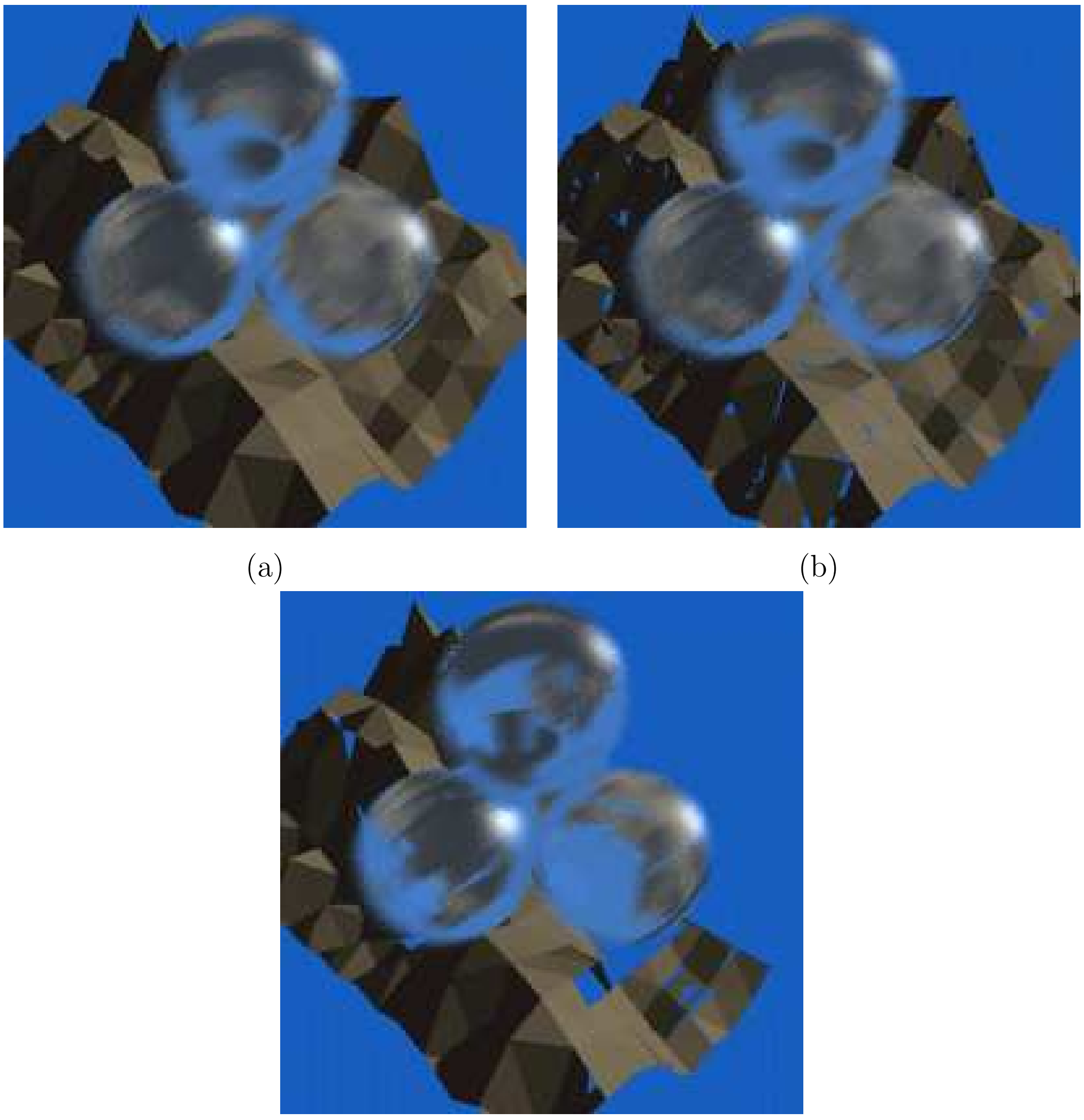
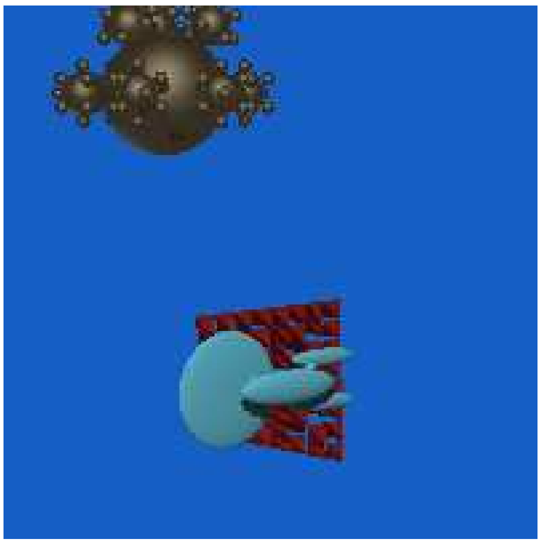
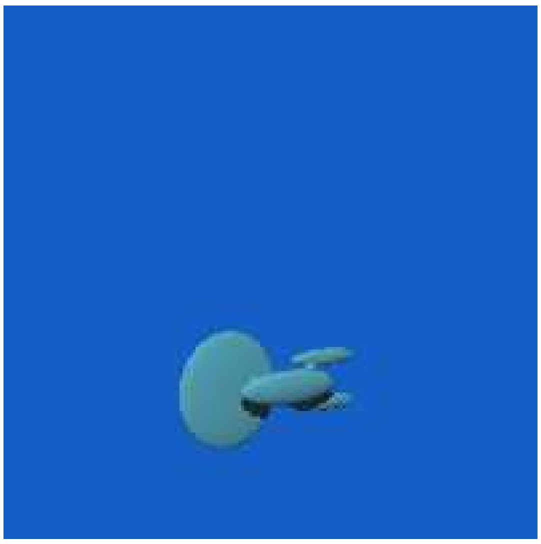


Figure 5.3: The same frame from “Mountains”; the background can be seen through the balls. Rendered using (a) exhaustive, (b) Glassner’s method, and (c) my method. Both (b) and (c) show how some objects were misclassified.


(a)


(b)

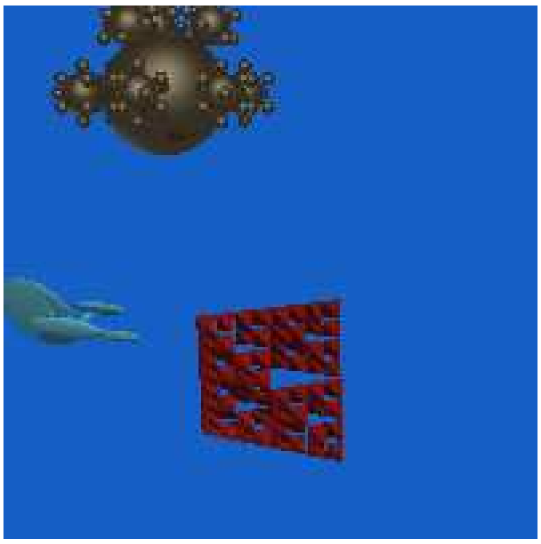

(c)

Figure 5.4: Frames from "Boldly", with motion blur turned off so the ship can be seen. (a) and (c) were rendered using the exhaustive method. (a) is the same frame as (c), but (c) was rendered using my method. Note that in (c), the static objects were completely misclassified.

Chapter 6

Discussion

This chapter discusses the results of the investigation that was the aim of this thesis. The investigation has looked at both the theoretical and empirical characteristics of both ray tracing efficiency schemes. In order to discuss the results, I will first present some criteria on which to compare the two methods.

The central issue of both Glassner's and my methods is to improve the efficiency of creating ray traced animations, by improving the spatial search method used to find the closest intersection of a ray and the scene. So most importantly, the fastest method is the best method. However, the are some issues regarding the meaning of "faster".

- *Run time.* The most intuitive meaning of "faster" is that the program runs in the shortest time. However, the true quality of this measurement is influenced by the quality of an implementation; for example, one method may run faster simply because all the functions were inlinned.

- *Number of comparisons.* The efficiency of these methods stem from the fact that they improve the spatial search; therefore, it is valuable to look at how much they improve just the spatial search phase. Looking at the number of comparisons needed in the search indicated the effectiveness of the search, irrespective of the optimisation of the program.

Ray tracing's approximation to the rendering equation is used to produce photorealistic images. Photorealistic renderers often employ complicated techniques to achieve life-like effects. Although it is beyond the scope of this project to produce an industry standard photorealistic renderer, it is important to look at how robustly the efficiency methods can handle these advanced features.

6.1 Theoretic comparison

This section will make a comparison of the efficiency methods based on the material presented in Chapter 2. Here is a point by point comparison of the major differences between Glassner's space time hierarchical bounding volumes and my space time ray classification method:

Glassner's	*Mine*
• Nested hierarchies of bounding volumes is a straight forward idea.	• Subdivision of ray space is initially a complex idea.
• Hard to implement, due to the complexity of surface/cell classification (took 3 weeks to implement).	• Easier implementation; bounding volume/beam classification quite simple (4 days to implement).
• Full hierarchy build as a pre-processing step.	• Only parts of hierarchy needed are build using lazy evaluation.
• Always divides along T axis.	• Only divides along T axis if necessary.
• Stores all parts of the hierarchy for the duration of the animation.	• Prunes unused parts of the hierarchy

These points indicate that space time ray classification holds many advantages. Glassner's method is difficult to implement because a surface/cell test is in general hard (Section 4.2). Ray classification only requires one algorithm for testing a bound against a beam, but Glassner's method requires a different surface/cell technique for each primitive type. This would make it hard to add complex photorealistic features, such as arbitrary object deformations, to the space time hierarchy scheme. An alternative might be to change the surface/cell test to, say, a bound/cell test. However, this may drastically change the characteristics of Glassner's method, since many more objects would be classified to a cell, creating much bigger hierarchies. I believe that the need for a surface/cell test is an implicit problem with Glassner's technique.

Space time ray classification builds a smaller hierarchy using lazy evaluation and smart T division, and theoretically has better memory requirements due to pruning. However, it is quite possible to use lazy evaluation to build Glassner's hierarchy. In fact, smart T division and pruning can also be used. Comparing space time ray classification with such an improved space time hierarchical technique would more accurately compare 3-space subdivision with ray space subdivision.

6.2 Empirical comparison

Table 5.5 shows that space time ray classification is an efficiency improvement over Glassner's technique, in all categories. For scenes with localised movement, ray classification was approximately 50% faster, and resulted in 62% less comparisons. For scenes with wide ranging movement, ray classification was 26–36% faster, with 63–75% fewer comparisons. Since Glassner's method did not complete for the "Night City" animation, we can expect that ray classification offers a substantial improvement for scenes where the camera moves significantly.

It is interesting to note that there is up to twice as much improvement in the number of comparisons compared with the improvement in rendering time. As would be expected, as spatial search techniques become more efficient, the proportion of time spent doing the search will decrease from the often quoted 95% [5].

Table 5.6 shows that ray classification uses *much* more memory than hierarchical bounding volumes; almost 1000 times more for the "Party-cool" animation. Ray classification divides each axis of $\mathcal{R}^3 \times T \times S^2$, which results in $2^6 = 64$ (or $2^5 = 32$ on nodes that use smart T division) children for each node. On the other hand, Glassner's method divides $\mathcal{R}^3 \times T$, which results in $2^4 = 16$ children per node. This indicates that ray classification inherently has a higher memory overhead, by orders of magnitude.

The "moving camera" animation "Night City" has 401 objects, and results in a hierarchy of 779×10^3 nodes; whereas "Party-cool", a wide ranging movement animation, has 10 objects (a factor of 40 difference) but results in a hierarchy of 1529×10^3 nodes! Ray classification seems to perform much better when there is little wide ranging movement. These results may also reflect the fact that "Night City" has mostly flat-edged objects, where "Part-cool" has all highly curved ones. However, "Mountains" also has a significantly curved component, but has a comparable memory usage to Glassner's method.

6.3 Comparison summary

The following statements can be made, based upon the last two sections:

- Space time ray classification is significantly more efficient than space time hierarchical bounding volumes, as Glassner presented it. This is expected from the theory, since my method uses lazy evaluation and smart T division, and ray classification explicitly takes ray direction into account when performing the spatial search.

- Ray classification has a much higher memory usage, due to the nature of its subdivision.
- The effectiveness of using Glassner's method in a photorealistic renderer is clouded by the potential complexity of surface/cell classification. Space-time ray classification offers a robust method for bound/beam classification, irrespective of the primitive type.

6.4 Quality of work

I would like to make some comments regarding the quality of the work in this project, and in the field. In 1992, Speer wrote [27]:

> . . . that research on ray tracing for image synthesis had accelerated in recent years: areas under active study now include ray tracing in data visualisation, ray tracing for radiosity and parallel ray tracing, in addition to more familiar ones like fast subdivision traversal, stochastic sampling and efficient intersection culling.

However since then, things have changed. Regarding research into ray traced animation, Haines wrote [25]

> Actually, I'm pretty happy to see people still researching this area, as I think we don't really know that much about how to do it well. Some research has been done, but there are so many other unexplored fronts that there hasn't been much concerted effort on ray tracing efficiency and animation since, like, Glassner [1] and Jevans [28].

Figure 6.1 shows the distribution of articles over the last 30 years, taken from the "Ray Tracing Bibliography" [29]. Research into ray tracing has steadily fallen since 1990. Very little work has gone into ray traced animations since Glassner's space time article; In fact, I could find no further research into actual space-time ray tracing since Glassner's article in 1988.

I believe Glassner's results with space time ray tracing, and my more efficient space time ray classification indicate that there is still much valuable work to be done in this area. Also, as far as I could determine, I was the first to suggest smart T dividing and to

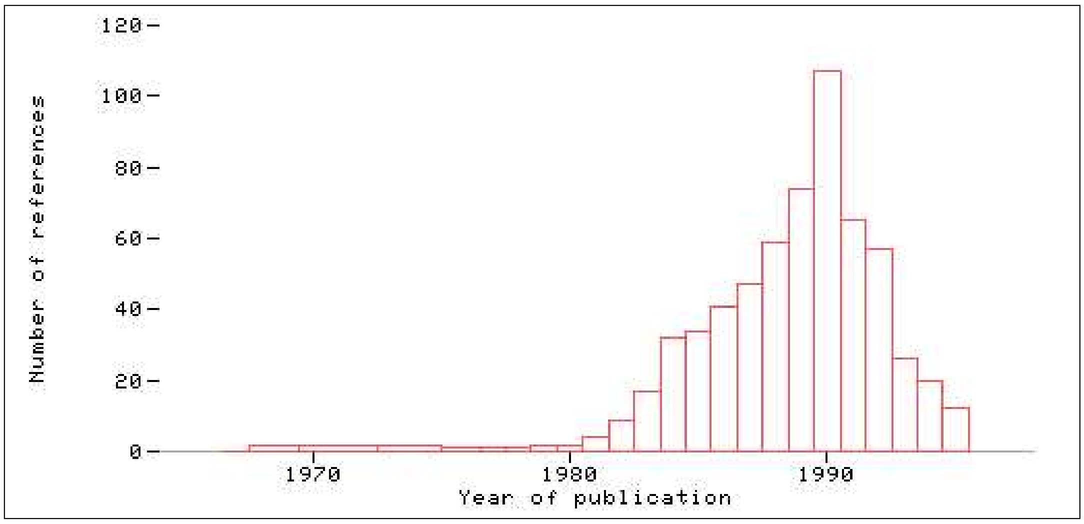


Figure 6.1: Distribution of publication dates for references on ray tracing.

use pruning for previous frames, and the first to implement space time ray classification (Arvo and Kirk suggested this but never implemented it).

One possible reason for the decline in research into ray tracing is that conventional techniques have become advanced enough to offer many of the photorealistic effects that ray tracing performs. It is my opinion, however, that there is still not enough known about efficiency in ray tracing for it to be dismissed as too inefficient.

6.5 Further work

The most immediate area for further work is to implement lazy evaluation, smart T division and pruning for Glassner's method. Glassner suggested lazy evaluation as further work in his article, but never followed it up. Lazy evaluation would remove the expensive preprocessing step that reduces the effectiveness of this method.

It is not entirely clear what features a renderer should support to cover a wide range of photorealistic features of industrial renderers. This is important, since adding sophisticated features may restrict the efficiency scheme that can be used.

Simiakakis [8] discussed parallel implementations of his Five Dimensional Subdivision. It would be worthwhile looking into how parallelism could be used for space time animation.

Chapter 7

Conclusion

The aim of this thesis was to investigate two methods for the efficient rendering of ray traced animations; Glassner's space time hierarchical bounding volume method and my space time ray classification. The focus of the investigation was on theoretical and implementation issues and empirical results.

Space time ray classification has many better features when compared with Glassner's method; including lazy evaluation, smart T division and past frame pruning. Using Glassner's method as-is, my technique renders up to 50% faster and uses around 70% less comparisons in the spatial search. However, Glassner's method could be extended to include lazy evaluation, smart T division and pruning.

Space time ray classification is much more memory demanding than Glassner's hierarchical bounding volumes, because of the branching factor of the hierarchy it creates. Even though my method uses two memory saving schemes, it still uses significantly more memory.

The range of advanced photorealistic features a renderer supports is an important factor for an efficiency scheme. Because of surface/cell classification, it is difficult to add features to Glassner's method. On the other hand, ray classification is largely independent on the complexity of features supported.

Apart for the initial article into space time ray tracing, very little research has been done into space time ray tracing, and I believe that I was the first to implement space time ray classification, even though Arvo and Kirk suggested it. Also, the research interest in ray tracing has continually fallen for the last five years, even though there is so much that is not understood about efficiency in animated ray tracing.

Bibliography

[1] Andrew S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, March 1988.

[2] James Arvo and David Kirk. Fast ray tracing by ray classification. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 55–64, July 1987.

[3] George Simiakakis. Accelerating ray tracing and directional subdivision. Master's thesis, University of East Anglia, 1992.

[4] J. T. Kajiya. The rendering equation. In David C. Evans and Rusell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, August 1986.

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

[6] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An introduction to ray tracing*, pages 201–262. Academic Press, 1989.

[7] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, volume 14, pages 110–116, July 1980.

[8] George Simiakakis. *Accelerating Ray Tracing with Directional Subdivision and Parallel Processing.* PhD thesis, University of East Anglia, October 1995.

[9] Andrew S. Glassner, editor. *An introduction to ray tracing.* Academic Press, 1989.

[10] James D Foley, Andries van Dam, Steven K Feiner, and John F. Huges. *Computer Graphics Principles and Practice.* Addison-Wesley Publishing Company, 1990.

[11] Pat Hanrahan. A survey of ray-surface intersecion algorithms. In Andrew S. Glassner, editor, *An introduction to ray tracing*, pages 79–119. Academic Press, 1989.

[12] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: theory and practice.* ACM Press, 1992.

[13] Walter F. Taylor. *The Geometry of Computer Graphics.* Wadsworth & Brooks/Cole, 1991.

[14] P. Shirley and C. Wang. Distribution ray tracing: Theory and practice. In *Eurographics Workshop on Rendering*, 1992.

[15] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In David C. Evans and Rusell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 269–278, August 1986.

[16] Jerrold E. Marsden and Anthony J. Tromba. *Vector Calculus.* W.H. Freeman and Company, thrid edition, 1988.

[17] Andrew Glassner. personal correspondence, 1996.

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

[19] L. Richard Speer. A new subdivision method for high-speed, memory efficient ray shooting. *Third Eurographics Workshop on Rendering*, pages 45–60, May 1992.

[20] Paul S. Heckbert. Writing a ray tracer. In Andrew S. Glassner, editor, *An introduction to ray tracing*, pages 263–293. Academic Press, 1989.

[21] Andrew S. Glassner. Surface physics for ray tracing. In Andrew S. Glassner, editor, *An introduction to ray tracing*, pages 121–160. Academic Press, 1989.

[22] David B. Kirk, editor. *Graphics Gems III.* Academic Press, San Diego, 1992.

[23] Andrew S. Glassner, editor. *Graphics Gems.* Academic Press, 1990.

[24] Eric Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 7(11):3–5, November 1987.

[25] Eric Haines. personal correspondence, 1996.

[26] Robert L. Cook. Stochastic sampling and distributed ray tracing. In Andrew S. Glassner, editor, *An introduction to ray tracing*, pages 161–199. Academic Press, 1989.

[27] L. Richard Speer. An updated cross-indexed guide to the ray-tracing literature. *Computer Graphics*, 26(1):41–72, January 1992.

[28] David A. Jevans. Object space temporal coherence for ray tracing. In *Proceedings of Graphics Interface '92*, pages 176–183, May 1992.

[29] Alf-Christian Achilles. On the World Wide Web at `http://bavi.unice.fr/Biblio-/Graphics/ray.html`.

Appendix A

Availability

This thesis was submitted as Technical Report C/TR96-08 to the Department of Computing at Macquarie University. A PostScript version of this report is available via anonymous FTP at

ftp://ftp.mpce.mq.edu.au/pub/comp/techreports/

The source code for the program and all the animations generated are stored at

ftp://ftp.mpce.mq.edu.au/pub/comp/src/

The animations are stored using AutoDesk's FLIC file format.

A compete listing of the source code follows.