Threaded octree structures for fast neighbor voxel processing in a parallel ray tracer

B.R. Naveen Chandra
Portland State University

Let us know how access to this document benefits you.
Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds
Part of the Electrical and Computer Engineering Commons

Recommended Citation

10.15760/etd.5859

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

Title: Threaded Octree Structures for Fast Neighbor Voxel Processing in a Parallel Ray Tracer.

APPROVED BY MEMBERS OF THE THESIS COMMITTEE:

Marek Perkowski, Chairman

Michael A. Aradiscio

Jagdish Ahuja

In the field of Computer Graphics, Ray Tracing has so far been the best algorithm for rendering of realistic three dimensional images created by mathematical models. Ray Tracing is also known for its very large computation times, where the computation depends on the picture resolution, the number of objects and the complexity of the scene.

In the last few years, several algorithms and special computer architectures have been proposed making Ray Tracing computationally less expensive. Parallelizing the algorithm has immensely speeded up the process.
Rendering images whose entities are built by mathematical models involves scene description, object definition and the rendering process. Ray Tracing implements the camera model which is very similar to human perception. The camera's view point is the focal point, and the film in the camera is the image plane. Ray Tracing computes the paths of light rays, and evaluates the objects described in the scene as seen from the view point. It computes reflection and refraction of light from the objects and the color of the object as perceived from the view point.

One of the computationally most expensive process in the algorithm is finding intersections of the ray and the objects described in the scene. Several methods for reducing these intersection calculations have been proposed. The method of using Bounding Volumes has given the best results in reducing the number of Ray-Object intersection calculations. In this method complex objects are bounded by simple bounding Volumes, or a number of objects can be bounded by one simple bounding volume. The intersection of the ray can be tested with this simple bounding volume. If there is an intersection with the bounding volume in the direction of the ray, only then does the intersection of the ray with the objects need to be considered.

An excellent scheme of creating bounding volumes is the Octree technique. It allows for dynamic subdivision of Object space into cubes of decreasing volume. These cubes also called as voxels (volume elements) can hold a certain number of objects described in the scene. Ray intersections are first tested with a voxel in the direction of the ray, and if an intersection exists, Ray-Object intersections are calculated with the objects contained in the voxel. One of the major problems here, is, finding the next neighboring voxel in the direction of the ray. Algorithms which have been proposed so far, address this problem by finding the voxel address from the top of the octree structure. This idea of finding the neighboring voxel from the top of the octree structure has been found to introduce a very significant computation overhead.
This thesis presents a new way of building Octree Structures called as Threaded Octree Structures. These Octree Structures not only minimize ray object intersections, but their special feature of internal threading, minimizes neighbor voxel processing, by reducing the overhead of searching the appropriate voxel from the top of the structure. The new algorithm uses pointer structures where each voxel in the Octree structure has pointers to its appropriate neighbors.

This new Octree Structure has shown significant improvements in speeding up the Ray Tracing process. Several images have been rendered, and speed improvements of upto 35% have been found. It was also found that as the complexity of the scene increases and the number of rays traced in the scene increases, there is a higher percentage speedup due to the new Threaded Octree Structure compared to the standard Octree Structure. For images with a number of objects less than 20, the improvement in speed is very limited and of the order of 5% to 10%. But since regular scenes contain a large number of objects in the order of thousands, the new proposed algorithm proves its usefulness in Ray Tracing. Previous research has found that 95% of the Ray Tracing computation time is spent on Ray-Object intersection calculations. This involved computing Ray direction, reflection, refraction sorting of objects and the actual intersection calculation itself. The proposed algorithm immensely reduces the sorting of objects since the objects are bounded by voxels. Sorting takes place within the voxel only for the objects assigned to that voxel. A maximum number of objects per voxel is initialized at the beginning of the Ray Tracing process. Computation in the new algorithm also involves finding the neighboring voxel, which is efficiently solved by threading the Octree Structure. The new algorithm also reduces the total number of intersection calculations. Experimental results have shown that this is of the order of 100000:1.

In this thesis, image space parallelism of the ray tracer was implemented on a
Cogent XTM parallel workstation using Cogent’s Kernel Linda parallel programming language. Nine T800 processors each with 4MB of memory were available. Processing times from one to nine processors have been registered. Five experimental images were developed, where each image was built to incorporate object complexity and a large total number of rays to be Ray Traced. A near linear speed up of the the Ray Tracer was achieved and the effective utilization of the processors was on an average of 82%.

This raytracer was also implemented on a Sun 386i uniprocessor machine and its results have been presented.

In all cases the new algorithm presented satisfactory speedup of the Ray Tracing algorithm over the previous algorithms presented in research papers.
THREADED OCTREE STRUCTURES FOR FAST NEIGHBOR VOXEL PROCESSING IN A PARALLEL RAY TRACER

by

B.R. NAVEEN CHANDRA

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1990
TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of B.R. Naveen Chandra presented July 5, 1990.

Marek Perkowski, Chairman

Michael A. Driscoll

Jagdish Ahuja

APPROVED:

Rolf Schaumann, Chairman, Department of Electrical Engineering

C. William Savery, Vice Provost for Graduate Studies and Research
ACKNOWLEDGMENTS

I would like thank my advisor, Dr. Marek Perkowski, for giving me all the encouragement and support in doing this research work. His dedication and hard working capacity has been a great source of inspiration for me.

In addition, I would like to thank Dr. Wm Leler of Cogent Research, Inc for helping me in using the parallel processing facilities at the Oregon Graduate Institute, without which this thesis could not have taken shape this early.

I would also like to thank the faculty, staff members, graduate students and my friends for their help, and my parents for their encouragement.

Naveen Chandra
July 5, 1990
Portland, Oregon
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>I  INTRODUCTION TO RAY TRACING</td>
<td>1</td>
</tr>
<tr>
<td>II THE RENDERING EQUATION</td>
<td>10</td>
</tr>
<tr>
<td>Parametric Equation Of A Line</td>
<td>10</td>
</tr>
<tr>
<td>Ray Surface Intersections</td>
<td>11</td>
</tr>
<tr>
<td>Ray Plane Intersection</td>
<td>11</td>
</tr>
<tr>
<td>Ray Polygon Intersection</td>
<td>12</td>
</tr>
<tr>
<td>Ray Sphere Intersection</td>
<td>15</td>
</tr>
<tr>
<td>Illumination</td>
<td>17</td>
</tr>
<tr>
<td>Shading</td>
<td>19</td>
</tr>
<tr>
<td>Specular Reflection Vector</td>
<td>19</td>
</tr>
<tr>
<td>Specular Refraction Vector</td>
<td>20</td>
</tr>
<tr>
<td>III SPACE SUBDIVISION FOR FAST RAY TRACING</td>
<td>24</td>
</tr>
<tr>
<td>Bounding Volumes</td>
<td>25</td>
</tr>
<tr>
<td>Octree Structures</td>
<td>26</td>
</tr>
<tr>
<td>Building Octrees</td>
<td>29</td>
</tr>
<tr>
<td>Ray Voxel Movement</td>
<td>32</td>
</tr>
<tr>
<td>Complexity of the above algorithm</td>
<td>38</td>
</tr>
<tr>
<td>New Algorithm</td>
<td>39</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Neighbouring Node Relationships : MAP I</td>
<td>39</td>
</tr>
<tr>
<td>2. Parallel Processing Time in seconds for Image of Figure 19.</td>
<td>76</td>
</tr>
<tr>
<td>2. Parallel Processing Time in seconds for Image of Figure 21.</td>
<td>77</td>
</tr>
<tr>
<td>2. Parallel Processing Time in seconds for Image of Figure 23.</td>
<td>78</td>
</tr>
<tr>
<td>3. Uniprocessor Result I</td>
<td>80</td>
</tr>
<tr>
<td>3. Uniprocessor Result II</td>
<td>81</td>
</tr>
<tr>
<td>FIGURE</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>1. A Camera Model</td>
<td>4</td>
</tr>
<tr>
<td>2. Ray Object Intersection</td>
<td>5</td>
</tr>
<tr>
<td>3. Reflection &amp; Refraction</td>
<td>7</td>
</tr>
<tr>
<td>4. Point Inside/Outside test for a polygon</td>
<td>13</td>
</tr>
<tr>
<td>5. Sphere</td>
<td>15</td>
</tr>
<tr>
<td>6. Diffuse and Specular Reflection</td>
<td>19</td>
</tr>
<tr>
<td>7. Refraction</td>
<td>21</td>
</tr>
<tr>
<td>8. Bounding Volumes</td>
<td>26</td>
</tr>
<tr>
<td>9. Object Bounding</td>
<td>27</td>
</tr>
<tr>
<td>10. Octree Hierarchy</td>
<td>28</td>
</tr>
<tr>
<td>11. Voxels</td>
<td>33</td>
</tr>
<tr>
<td>12. Ray - Voxel Boundary Intersections</td>
<td>34</td>
</tr>
<tr>
<td>13. Voxel Movement</td>
<td>35</td>
</tr>
<tr>
<td>14. Massively Threaded Octree Structure</td>
<td>42</td>
</tr>
<tr>
<td>14. Pointer Mappings</td>
<td>45</td>
</tr>
<tr>
<td>16. Shared versus Distributed Systems</td>
<td>50</td>
</tr>
<tr>
<td>17. Linda Messages</td>
<td>52</td>
</tr>
<tr>
<td>18. A Processor Farm</td>
<td>54</td>
</tr>
<tr>
<td>19. IMAGE 1</td>
<td>73</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION TO RAY TRACING

The best means of understanding theory is by implementing models. Models by themselves can be practical or imaginary. In most areas of Design and Research, constructing a model can be cost prohibitive both in the sense of time and money. Model failures and redesigns heavily add up to this cost. By simulating these models, one can decrease the cost involved and considerably increase the efficiency of the design and test cycle.

Computer Graphics provides the cheapest means of simulation for most areas of research or design, especially in the area of Scientific Visualization where models can be highly imaginative. Hence Computer Graphics is finding its way into areas like Complex Molecular Modelling for Chemistry and Bio-Medicine, Mechanical and Architectural Computer Aided Design and Presentation Graphics in business and advertisements, as well as for animation in the film industry.

A good visual presentation of a model always emphasizes the subject in discussion. This visual model can be presented on a screen of a computer, or by hard copy devices such as a plotter or printer. Models in two-dimensional or three-dimensional form can be rendered onto a computer screen by using various Computer Graphics techniques. There are several algorithms for rendering such images. Methods such as Depth-Buffer, Z-Buffer, Scan-Line, Area Subdivision algorithms [17], and shadow algorithms are used to generate three dimensional images.

Natural physical properties such as three dimensional view, surface shade, hidden surfaces, specular reflection and refraction effects are very difficult to model. The
above methods are not able to create all of these properties. Hence rendering very high quality images needs an algorithm which can support most of the above physical properties. Ray Tracing is the most efficient algorithm which can be used as a tool to synthesize very realistic images [2]. It implements a natural phenomenon of viewing objects and it effectively models all the optical effects in a single comprehensive algorithm.

Ray Tracing is a non-projective method. Most other standard hidden surface algorithms are projective methods: the surface elements of the objects are projected onto the image plane and a visibility calculation is performed, based on a depth sort prior to projection for all surface elements of the object (list priority algorithms), a depth sort for every pixel (Z-buffer algorithms), or a depth sort for each scan-line segment (scan-line algorithms). These algorithms are well described in Foley and Van Dam [17]. The projective methods are restricted to solve the hidden surface problem in one direction, and can strongly benefit from efficient sorting techniques. In the multidirectional case of ray tracing the use of coherence and sorting is not so straightforward.

Ray tracing has a history of use in CAD applications. This technique has shown up periodically in several CAD efforts because of its ability to generate images for a variety of geometric forms. There is also an assortment of other engineering analyses that take advantage of this technique. Applications in radar and antenna analysis have been suggested [27].

There is a great need for rapid production of images and the determination of mass properties for mechanical engineering computer-aided design [28]. Geometric modeling efforts are largely focused on solid modeling of mechanical parts. Constructive Solid Geometry (CSG) is used to generate the Geometric database. During the design process, the engineer requires rapid and abundant feedback concerning the
effects of the design decisions being made. Current CSG rendering programs are not fast enough to be incorporated in an interactive design process. "Quick-and-dirty" imaging has been achieved by approximating the solid primitives with faceted polygonal surfaces. This gives dramatic speed improvements. However, even large numbers of polygons do not describe a smooth surface exactly. For accurate imaging, the exact surface types must be used. Also, the quick polygonal-approximation schemes do not lend themselves to the variety of special display effects that ray tracing does.

Ray Tracing easily models the primitive objects such as polygons, spheres, and general quadratic surfaces such as cylinders, ellipsoids, paraboloids, as well as polyhedral volumes, the Parametric surfaces such as cubic splines, etc. To simulate these objects on a raster device, the algorithm implements a camera model which is very similar to that of a human perception. The camera’s view point is the focal point, and the film in the camera becomes the image plane. The figure below shows the camera model implemented by Ray Tracing.

Ray Tracing, as its name suggests, computes the paths of rays in a scene and evaluates the objects as to be seen from a viewpoint. In a typical environment, however, there is an infinite number of rays generated from a light source. These rays reach the eye point after several reflections from the objects. Modelling such infinite ray paths is impossible for any algorithm. We observe that only a small percentage of light rays: i.e., the light rays from the light source and the rays reflected from objects, will reach the eye point.

The rest of the rays which do not reach the eye point are considered lost in space, since these rays do not contribute to the perception of a scene. Some of the rays die down as the energy of the individual light rays falls down. Ray Tracing implements a smart method, in which the path of the rays reaching the eye are traced backwards, from the eye towards the light source.
The screen consists of pixels (picture elements) in the order of a million. Implementing the camera model, a ray is traced from the eye point through each of these pixels and into the environment being perceived. This ray intersects the objects defined in the environment, and causes reflections or refractions depending on the surface properties of each object. These reflected or refracted rays are further traced in the environment, looking for more intersections with other objects. An intersection tree is hence created for each ray, traced from the eye point towards each pixel. At every intersection point the amount of light reflected by the object is calculated with respect to the objects surface properties, the properties of the light source and the properties of other objects in the environment.

![Diagram of a Camera Model](image)

**Figure 1.** A Camera Model

This algorithm has been derived from perception of objects in real life. In reality, light rays from the light source strike the objects and branch out due to reflections or refractions. When these objects are perceived through a viewing plane, we observe
that, only those rays passing through the viewing plane reach the eye, and the rest of the light rays are lost in space.

![Diagram](image)

**Figure 2.** Ray Object Intersection

It is impossible to compute such infinite number of rays, and hence Ray Tracing implies a smart method of viewing the scene, only in the direction from the observers view point towards the light source.

Figure 2 shows an intersection tree for a single ray from the view point. The scene consists of three spheres and a cuboid. Observe that three new rays are traced at each intersection point. For ray R0 from the view point which intersects sphere 1, R1
is the reflection of the ray, T1 is the refraction of the ray, L1 is the ray from the light source. Reflection R1 and refraction T1 further probe into the scene looking for possible intersections with other objects. Therefore each of these rays branches out into a tree of rays probing into the environment. A limit is attached to the depth of this tree, or the upper bound of the contribution of any node of the intersection tree to the final color of the sample pixel is approximated. By establishing a cutoff contribution threshold, one can Adaptively Control the depth of the Ray Tracing Process. Thus the total light intensity at the the root of the tree is evaluated, with respect to the intensity contributions from each intersection point. The total intensity at each pixel is the recursively calculated sum of the intensity contributions at all nodes of the intersection tree.

The surface properties of the object and the properties of the light source determine the intensity at any intersection point. There are two basic surfaces defined: Opaque and Transparent. Opaque objects have either diffuse or reflective surfaces. The perceived intensity of reflected light from a diffuse surface is determined by the surface color and the light source illuminating the object. It is independent of the viewing angle. The perceived intensity of reflected light from a perfectly reflective surface, is a function of the angle between the reflected light from the light source and the viewing angle. Most objects have a percentage of both diffuse and reflective properties as seen in a natural environment.

The computation time for the intensity at each pixel is a linear function of

- The number of objects in the scene,
- The number of light sources,
- The depth of the intersection tree,

These factors, including the resolution of the picture, make Ray Tracing the most expensive synthetic image generator. The classic paper on Ray Tracing by Turner Whitted [2] indicates that 95% of the image generation time is spent on
evaluating the ray object intersections.

Figure 3. Reflection & Refraction

The efficiency of intersection routines makes a significant impact on the entire Ray Tracing algorithm. Proper floating point calculations become an important part since their quality determines the quality of the image generated.

Certain objects like polygons, spheres, cylinders and cones have simple intersection tests when compared to polyhedrons with many faces, or bi-polynomial parametric surfaces like Beta-Splines [17].

These problems have driven new research into developing faster algorithms and embedding these algorithms in hardware.

The idea of using bounding volumes on the objects has been suggested by
many researchers [1],[3],[4]. Neighboring objects are grouped into bounding volumes, which reduces the number of intersections on the objects. The intersection of the ray with the bounding volume is first tested. If the intersection exists, then the ray is further tested for intersection with the objects bounded by this volume. This obviously reduces the total number of intersections to be computed since the ray-object intersections are tested only for objects contained within that bounding volume. There are, however, other problems involving the bounding volumes, such as the maintenance of a hierarchy of the bounding volumes, and the calculation of the rays traversing between these volumes and intersecting them. Hence there is a great need to cut down on time spent on the intersection routines. This thesis presents a new method into looking at Bounding Volumes, in order to reduce the number of intersections within bounding volumes.

The main objective of this thesis was to build an efficient Ray Tracer incorporating the octree method of object space division. The octree technique is a type of object space subdivision. It allows for dynamic subdivision of object space into cubes of decreasing volume. The new algorithm proposed in this thesis effectively cuts down the time taken for a ray to traverse in the object space describing the scene. It implements pointer structures to point the ray to neighboring object bounding volumes in the direction of the ray. Comparisons with previously implemented algorithms by Glassner[1], Kobayashi[3] and others have been presented. The method presented in this thesis is simpler and more effective in its ability to be implemented on both uniprocessor and multiprocessor machines compared with the other algorithms.

The Ray Tracer was built using the 'C' language and implemented on a parallel machine, the Cogent XTM, using the Linda parallel programming. Tests were conducted in parallelizing the Ray Tracer on one to nine processors on the XTM's servers. A near linear speed was achieved. The results are presented in Chapter 5 of this thesis.
Some principles of Ray Tracing were introduced in this chapter. The subject of Image Rendering is discussed in the next chapter. The third chapter deals with Octree Structures, their implementation in the Ray Tracer, and their new features proposed in this thesis. Parallelization of the Ray Tracer and its implementation on the Cogent XTM Parallel Processors using Linda, is described in the fourth chapter. Chapter 5 presents few rendered images and their benchmarks. Chapters 6 includes conclusions and certain ideas for future work.
CHAPTER II

THE RENDERING EQUATION

The heart of any ray tracing package is the set of ray intersection routines. No matter what type of lighting models, texture mappings, space subdivision techniques, anti-aliasing schemes, or other elaborations of the ray tracing algorithm are desired, there is always the need to find the intersection point of a ray and an object.

PARAMETRIC EQUATION OF A LINE

Light rays can easily be represented by straight line equations, where the length of the line is an important parameter [21]. For each ray, a ray origin point is given, and its direction is determined from its destination point. The parametric equation of a line is given as:

\[ P_t = P_0 + (P_1 - P_0)t \]  \hspace{1cm} [1]

where \( P_0 \) is the origin, \( P_1 \) is the destination, \( t \) is a parameter from 0 to 1, and \( P_t \) is the point on the line.

Hence the point can be evaluated in 3D space as

\[ X_t = X_0 + (X_1 - X_0) * t \]  \hspace{1cm} [1a]
\[ Y_t = Y_0 + (Y_1 - Y_0) * t \]  \hspace{1cm} [1b]
\[ Z_t = Z_0 + (Z_1 - Z_0) * t \]  \hspace{1cm} [1c]

The direction of the line is represented as \( dx \), \( dy \), \( dz \), where

\[ dx = (X_1 - X_0) \]  \hspace{1cm} [1d]
\[ dy = (Y_1 - Y_0) \]  \hspace{1cm} [1e]
\[ dz = (Z_1 - Z_0) \] ...... [1f]

The direction of the ray from the viewpoint is evaluated by having the viewpoint as the origin and the destination point as the pixel on the screen. The picture plane is traversed scanline by scanline from left to right and from top to bottom.

RAY SURFACE INTERSECTIONS

Ray Plane intersection

Calculating intersection points on the objects, with respect to the ray is an important process of the entire Ray Tracing procedure [18]. The equation of a plane is given by:

\[ Ax + By + Cz + D = 0 \] ...... [2]

Where A,B,C represent the coordinates of the normal vector to the plane, x,y,z represents a point on the plane and D is the distance of the plane from the origin of its coordinate system.

Using the parametric equation of the line and the equation of the plane, the intersection of the line with the plane is calculated. Substituting the line equation in the plane equation,

\[ A(X_0 + X_d \cdot t) + B(Y_0 + Y_d \cdot t) + C(Z_0 + Z_d \cdot t) + D = 0 \] ...... [3]

Solving for \( t \):

\[ t = \frac{-(A*X_0 + B*Y_0 + C*Z_0 + D)}{A*X_d + B*Y_d + C*Z_d} \] ...... [4]

In vector notation, this equation is:

\[ t = \frac{-(P_n \cdot R_o + D)}{P_n \cdot R_d} \] ...... [5]

Where \( P_n \) is the vector normal to the plane, \( R_o \) the ray origin , and \( R_d \) the direction of the ray. If \( P_n \cdot R_d = 0 \) then the ray is parallel to the plane and no intersection occurs. A ray could be in the same plane, but this case is irrelevant in practice; hitting a polygon
edge-on has no effect on rendering. Also, if this dot product is greater than zero, the normal of the plane is pointing away from the ray. The sign of the normal vector \( P_n \) may be adjusted depending on its relationship with the direction \( R_d \). The sign of the normal vector should be reversed in order to point back toward the ray origin.

After obtaining \( t \) the intersection point is calculated as:

\[
\begin{bmatrix}
x_i \\
y_i \\
z_i
\end{bmatrix} = \begin{bmatrix}
x_0 + x_d \ast t \\
y_0 + y_d \ast t \\
z_0 + z_d \ast t
\end{bmatrix} \quad \text{[6]}
\]

**Ray Polygon Intersection**

Every polygon can be defined to lie on a plane. The polygon is defined by the coordinates of all of its vertices. The plane equation can then be derived from these vertices, by deriving the normal to the plane from the vector cross product of two adjacent sides of the polygon.

First the intersection point of the ray with the plane is derived, as shown in the previous section. This intersection point has to be tested to determine if it is inside or outside the polygon. There are several methods used for doing this inside/outside test. The method implemented here is that of shooting a ray in an arbitrary direction from the intersection point and counting the number of line segments crossed. If the number of crossings is odd, the point is inside the polygon; else it is outside. This is also known as the Jordan Curve theorem [18]. Special cases where the test ray intersects a vertex in the polygon is taken care of. The polygon is defined as a set of \( N \) points:

\[
\begin{bmatrix}
X_n \\
Y_n \\
Z_n
\end{bmatrix}, \text{ where } n = (0,1,\ldots,N-1) \quad \text{[7]}
\]

The first step is to project the polygon onto a two dimensional plane. In this plane all points are specified by pairs \((U,V)\), so that for each \([X \ Y \ Z]\) coordinate a \((U,V)\) pair exists, leaving the topology unchanged. This orientation is represented in Figure 4.

One method would be to rotate the polygon around some axis such that the normal is parallel to an other axis. The two remaining axes could be used to generate the \((U,V)\) pairs. This method is inefficient, since a rotation matrix must be generated
and stored for each polygon, and a matrix multiply must be performed for each coordinate.

\[ \text{Figure 4. } \text{Point Inside/Outside test for a polygon} \]

A simpler way would be to remove one of the \([X \ Y \ Z]\) coordinates and use the other two coordinates. This action projects the polygon onto the plane defined by the two chosen coordinates. The area of the polygon is not preserved, but the topology stays the same. Choosing which coordinate to remove is defined as follows: delete the coordinate whose corresponding plane equation value is of the greatest magnitude. Once the polygon has been projected upon a plane, the inside/outside test of the intersection point is fairly simple.

The algorithm used for the above procedure is given below:

1. Project the \(NV\) vertices \([X_n \ Y_n \ Z_n]\), where \(n = 0\) to \(NV-1\), onto the dominant coordinates plane, creating a list of vertices \((U_n, V_n)\).

2. Translate the \((U,V)\) polygon so that the intersection point is the origin. Call these points \((U'_n, V'_n)\).
3. Set the number of crossings NC to zero.

4. Set the sign holder SH as a function of \( V'_o \), which is the \( V' \) value of the first vertex of the first edge. SH is set to -1 if \( V'_o \) is negative, and is set to 1 if \( V'_o \) is zero or positive.

FOR EACH EDGE of the Polygon formed by points \((U'_a, V'_a)\) and \((U'_b, V'_b)\)
where \( a = 0 \) to NV-1, \( b = (a+1) \mod \) NV: Set the sign holder NSH to -1 if \( V'_b \) is negative, and to +1 if \( V'_b \) is zero or positive.

If SH is not equal to NSH: {

If \( U'_a \) is positive and \( U'_b \) is positive
then line must cross +\( U' \), \( NC = NC + 1 \);

Else if either \( U'_a \) is positive and \( U'_b \) is positive
then line crosses \( U' \) axis. the Compute \( U' \) axis intersection;

If \( U'_a - ((U'_b - U'_a) / (V'_b - V'_a)) \cdot V'_a > 0 \)
then line must cross +\( U' \), \( NC = NC + 1 \);

} 

Set SH = NSH

NEXT EDGE

If NC is odd, the point is inside the polygons, else it is outside. This method is highly efficient because most edges can be trivially rejected or accepted [17]. Only when the edge extends from diagonally opposite quadrants does any serious calculations need to be performed.
Ray Sphere Intersection

The general quadratic equation of a sphere is given as

\[ x^2 + y^2 + z^2 = R^2 \quad \ldots \quad [8] \]

where \( R \) is the radius of the sphere, and \( X, Y, Z \) are coordinates of a point on the surface of the sphere. The sphere is assumed to be centered at the origin. The equation for a sphere of an arbitrary center is

\[ (X - X_c)^2 + (Y - Y_c)^2 + (Z - Z_c)^2 = R^2 \quad \ldots \quad [9] \]

where \( X_c, Y_c, \) and \( Z_c \) are the coordinates of the center of the sphere.

![Figure 5. Sphere](image)

The sphere's surface is expressed as an implicit equation. In this form points on the surface cannot be directly generated. Instead, each point \([ X, Y, Z ]\) can be tested by the implicit equation; if it fulfills the equation's conditions, the point is on the surface. To solve the intersection problem, the ray equation is substituted into the sphere equation and the result is solved for \( t \). If \( X_0 \) is the origin of the ray, and \( X_d \) is the direction, then the equations are:
\[ X = X_0 + X_d \ast t \quad \text{[10a]} \]
\[ Y = Y_0 + Y_d \ast t \quad \text{[10b]} \]
\[ Z = Z_0 + Z_d \ast t \quad \text{[10c]} \]

Substituting the equations into the sphere’s equation:
\[
(X_0 + X_d \ast t - X_c)^2 + (Y_0 + Y_d \ast t - Y_c)^2 + (Z_0 + Z_d \ast t - Z_c)^2 = R^2 \quad \text{[11]} 
\]

This simplifies to
\[
A \ast t^2 + B \ast t + C = 0 \quad \text{[12]} 
\]

where
\[
A = X_d^2 + Y_d^2 + Z_d^2 = 1 \quad \text{[13]} 
\]
\[
B = 2 \ast (X_d \ast (X_0 - X_c) + Y_d \ast (Y_0 - Y_c) + Z_d \ast (Z_0 - Z_c)) \quad \text{[14]} 
\]
\[
C = (X_0 - X_c)^2 + (Y_0 - Y_c)^2 + (Z_0 - Z_c)^2 - R^2 \quad \text{[15]} 
\]

Coefficient A is 1 since the direction of the ray is normalized. The equation is quadratic and the solution for t is:
\[
t_0 = \frac{-B - \sqrt{B^2 - 4 \ast C}}{2} \quad \text{[16a]} 
\]
\[
t_1 = \frac{-B + \sqrt{B^2 - 4 \ast C}}{2} \quad \text{[16b]} 
\]

When the discriminant is negative (the part in the sqrt function), the line misses the sphere. The smaller positive real root is taken as the closest intersection point on the ray. If no such point exists, then the ray misses the sphere. Once the distance t is found, the actual intersection point is:
\[
[x_i, y_i, z_i] = [X_0 + X_d \ast t, Y_0 + Y_d \ast t, Z_0 + Z_d \ast t] \quad \text{[17]} 
\]

The normal at this intersection point is given as
\[
r_{\text{normal}} = \left[ \frac{(x_i - X_c)}{R}, \frac{(y_i - Y_c)}{R}, \frac{(z_i - Z_c)}{R} \right] 
\quad \text{[18]} 
\]

For transparent spheres where the ray may originate inside the sphere and hit the internal walls, the normal should be negated so that it points back towards the ray.
ILLUMINATION

In order to generate realistic images we need to understand how light behaves at the surfaces of objects. There have been several illumination models proposed. The role of any illumination model is to determine how much light is reflected to the viewer from a visible point on a surface, as a function of the light source direction and strength, viewer position, surface orientation, and surface properties.

We can define three simple types of surfaces of objects. Diffuse, Reflective, and Transparent [17]. Dull (matte) surfaces exhibit diffuse reflection, scattering light equally in all directions, so that the surfaces appear to have the same brightness from all viewing angles. For such surfaces, Lambert's cosine law [18] relates the amount of the reflected light to the cosine of the angle $\theta$. This angle is between the normal at the point on the surface to the direction of the light source of intensity $I_p$.

The diffuse illumination is given as:

$$I_d = I_p Kd \cos \theta \quad \ldots \quad [19]$$

where $Kd$ is a constant for the object's light diffusing property, and varies between 0 and 1. $\cos \theta$ can be defined as a dot product of the two normalized vectors $L$ & $N$, where $L$ is the vector in the direction of the point light source, and $N$ is the normal vector to the surface.

$$I_d = I_p Kd \ (L \cdot N) \quad \ldots \quad [20]$$

Specular reflection can be observed on any shiny surface. For example, illuminating an apple with a bright light, we observe a highlight on its surface, which is due to the specular reflection. The light reflected from the rest of the surface is due to the diffuse reflection. We observe that the highlight is not the same as the color of the apple, but it is the color of the light. In the case of white light, we observe a white highlight. This indicates that, the specular reflection is a function of the light source and the view point, rather than a function of the light source and the surface normal.
All shiny surfaces reflect light only in the direction for which the "angle of incidence" and the "angle of reflection" are equal. This is true especially for perfect reflectors such as a mirror. For the imperfect reflectors such as an apple, the intensity of the reflected light falls off sharply as the viewing angle increases. This rapid fall off is modelled as $\cos \alpha^n$, where $n$ varies from 1 to 200, depending on the surface. For a perfect reflector the value of $n$ is infinite. This model is not a fundamental model of the specular reflection process, but is a reasonable approximation based on empirical observation.

The Specular reflection is given as:

$$I_r = Ip K_{sr} \cos \alpha^n$$

... [21a]

The Specular refraction is given as:

$$I_r = Ip K_{st} \cos \alpha^n$$

... [21b]

where $K_{sr}$ and $K_{st}$ are the specular constant factors of the object, and vary between 0 and 1. $\cos \alpha$ can be defined as the dot product of the two normalized vectors $R \& N$.

$$I_r = Ip K_{sr} (R \cdot N)^n$$

... [22a]

$$I_r = Ip K_{st} (R \cdot N)^n$$

... [22b]

In any real environment, there is a certain amount of ambient light falling on the objects. This ambient light is in fact produced by the inter-reflections within the environment. It is therefore appropriate to model a degree of the ambient light into the lighting environment in order to generate a more realistic image. Defining $I_a$ to account for the ambient light, a constant $K_a$ is introduced to indicate how much of the ambient light is reflected from the surface.

$$I = I_a K_a \quad \text{where} \quad K_a \text{ varies from 0 to 1.}$$

... [23]
SHADING

The above illumination model is used for finding the intensity of the reflected light at every intersection point. The shading used here is very similar to that defined by Whitted [2] which is originally based on the Phong shading model [17]. The intensity of light on the surface is calculated for every intersection point, as:

\[ I_r = K_a \text{Object.r} + K_d \sum_{l=1}^{l=m} L p(L \cdot N) + K_{sr} \sum_{l=1}^{l=m} L p K_{sr} (R \cdot N)^s + K_{st} \sum_{l=1}^{l=m} L p K_{st} (R \cdot N)^s \ldots [24] \]

\[ I_g = K_a \text{Object.g} + K_d \sum_{l=1}^{l=m} L p(L \cdot N) + K_{sr} \sum_{l=1}^{l=m} L p K_{sr} (R \cdot N)^s + K_{st} \sum_{l=1}^{l=m} L p K_{st} (R \cdot N)^s \ldots [25] \]

\[ I_b = K_a \text{Object.b} + K_d \sum_{l=1}^{l=m} L p(L \cdot N) + K_{sr} \sum_{l=1}^{l=m} L p K_{sr} (R \cdot N)^s + K_{st} \sum_{l=1}^{l=m} L p K_{st} (R \cdot N)^s \ldots [26] \]

where \( r, g \) and \( b \) represent the red, green and blue primary components of a color, and Object.r, Object.g & Object.b are the r,g,b color ratios of the object.

SPECULAR REFLECTION VECTOR

\[ \text{Figure 6. Diffuse and Specular Reflection} \]

Let \( I \) be the light vector, \( R \) be the reflection vector and \( N \) be the normal vector to the surface at the intersection point, as shown in Figure 6.

There are several ways to find \( R \), an algebraic solution will be used here to find \( R \). We know that these three vectors lie on the same plane; thus the reflected ray is a linear combination of the incident ray and the normal. Also the angle of incidence is equal to the angle of reflection.
\[ R = \alpha I + \beta N \] \hspace{1cm} \text{[27]} \\
\[ \theta = \theta \] \hspace{1cm} \text{[28]} \\

From figure 6, one finds that

\[ \cos \theta_i = -I \cdot N \text{ and } \cos \theta_r = N \cdot R \] \hspace{1cm} \text{[29]} \\

Substituting these cosine angles for equation 28, we have

\[ \cos \theta_i = \cos \theta_r \] \hspace{1cm} \text{[30]} \\

From equation 29, it can be derived that

\[ -I \cdot N = N \cdot R \] \hspace{1cm} \text{[31]} \\

Substituting for \( R \) from equation 27,

\[ -I \cdot N = N \cdot (\alpha I + \beta N) \] \hspace{1cm} \text{[32]} \\

Evaluating we find that \( \alpha = 1 \) and \( \beta = -2(N \cdot I) \)

So the complete formula for the direction of a specularly reflected ray is

\[ R = I - 2(N \cdot I) \times N \] \hspace{1cm} \text{[33]} \\

**SPECULAR REFRACTION VECTOR**

Let \( I \) be the incoming light vector, \( T \) be the transmitted vector and \( N \) be the normal vector to the surface at the intersection point (see figure 7). Due to the refraction, the angle of incidence is not equal to the angle of refraction, and is given by the Snell's law

\[ \frac{\sin(\theta_i)}{\sin(\theta_2)} = \eta_{21} = \frac{\eta_2}{\eta_1} \]

or \[ \frac{S_i}{S_2} = \eta \] \hspace{1cm} \text{[34]} \\

where:

\( \eta_1 \) is the index of refraction of medium 1 with respect to vacuum,
$\eta_2$ is the index of refraction of medium 2 with respect to vacuum,

$\eta_{21}$ is the index of refraction of medium 2 with respect to medium 1.

The index of refraction is dependent on the wavelength of the incoming light. This can be perceived when light gets refracted in a prism, we observe a spectrum of light, as different wavelengths are refracted by different amounts.

Here too the three vectors lie on the same plane; thus the refracted ray $T$ is a linear combination of the incident ray $I$ and the normal vector $N$

$$T = \alpha \, I + \beta \, N \quad \text{[35]}$$

While calculating $T$, an important phenomenon of light reflection through transparent materials called Total Internal Reflection has to be taken into consideration. This physical phenomenon occurs when the light tries to pass from a dense medium to a less dense medium at too shallow an angle. The result is that the light glances off the interface between the media, and is in effect specularly reflected instead of transmitted. Instead of finding the critical angle beyond which this phenomenon occurs, one can
just detect when this Total Internal Reflection occurs.

An algebraic solution for $T$ has been implemented here. From Figure 7, we have

$$\cos(\theta_i) = C_i = (N, -T) \quad \ldots \quad [36]$$

$$\cos(\theta_t) = C_t = (-N, T) \quad \ldots \quad [37]$$

Squaring both sides of the Snell's law:

$$\frac{S_s^2}{S_i^2} = \eta_i^2 \quad \ldots \quad [38]$$

$$S_i^2 \eta_i^2 = S_s^2 \quad \ldots \quad [39]$$

Since $S_s^2 + C_s^2 = 1$, replace this with

$$(1 - C_i^2) \eta_i^2 = (1 - C_i^2) \quad \ldots \quad [40]$$

which can be rewritten as:

$$(1 - C_i^2) \eta_i^2 - 1 = (1 - C_i^2) \quad \ldots \quad [41a]$$

$$(1 - C_i^2) \eta_i^2 - 1 = [\alpha(N \cdot T)^2] \quad \ldots \quad [41b]$$

$$(1 - C_i^2) \eta_i^2 - 1 = [\alpha(N \cdot (\alpha I + \beta T)]^2 \quad \ldots \quad [41c]$$

$$(1 - C_i^2) \eta_i^2 - 1 = [\alpha(-N \cdot I) + \beta(-N \cdot N)]^2 \quad \ldots \quad [41d]$$

$$(1 - C_i^2) \eta_i^2 - 1 = [\alpha C_i - \beta]^2 \quad \ldots \quad [41e]$$

The last step is justified by noting that $(N \cdot N) = 1$ (since $|N| = 1$). Equation [41a] is the condition on $\alpha$ and $\beta$. Since the new vector $T$ should have unit length, the condition can be stated as:

$$1 = T \cdot T \quad \ldots \quad [42a]$$

$$1 = (\alpha I + \beta N) \cdot (\alpha I + \beta N) \quad \ldots \quad [42b]$$

$$1 = \alpha^2(I \cdot I) + 2\alpha\beta(I \cdot N) + \beta^2(N \cdot N) \quad \ldots \quad [42c]$$

$$1 = \alpha^2 - 2\alpha\beta C_i + \beta^2 \quad \ldots \quad [42d]$$

where again the fact that $(I \cdot I) = (N \cdot N) = 1$ is used, and $(I \cdot N)$ is replaced by $-C_i$ from equation [36]. The results of these two derivations can be written as:

$$(1 - C_i^2) \eta_i^2 - 1 = [\alpha C_i - \beta]^2 \quad \ldots \quad [43a]$$
\[ 1 = \alpha^2 - 2\alpha \beta C_i + \beta^2 \] \hspace{1cm} \ldots \ldots \ [43b]

Four values of \( \alpha \) and \( \beta \) can be recognized for the square roots derived by solving the above simultaneous equations.

\[ w = \eta_{ir} C_i \] \hspace{1cm} \ldots \ldots \ [44a]
\[ v = \sqrt{1 + \eta_{ir}^2 (C_i^2 - 1)} \] \hspace{1cm} \ldots \ldots \ [44b]

This can be written as:

\[ \alpha_1 = \eta_{ir}, \quad \beta_1 = w - v \] \hspace{1cm} \ldots \ldots \ [45a]
\[ \alpha_2 = \eta_{ir}, \quad \beta_2 = w + v \] \hspace{1cm} \ldots \ldots \ [45a]
\[ \alpha_3 = -\eta_{ir}, \quad \beta_3 = -w + v \] \hspace{1cm} \ldots \ldots \ [45a]
\[ \alpha_4 = -\eta_{ir}, \quad \beta_4 = -w - v \] \hspace{1cm} \ldots \ldots \ [45a]

The first set of \( \alpha \) and \( \beta \) correspond to the \( T \) vector; the others represent reflections of that vector into the other three quadrants formed by the normal and the surface’s tangent. Thus the final formula for \( T \) is:

\[ T = \eta_{ir} I + (\eta_{ir} C_i - \sqrt{1 + \eta_{ir}^2 (C_i^2 - 1)}) N \] \hspace{1cm} \ldots \ldots \ [41]

The expression under the square root is negative, leading to an imaginary solution, if the Total Internal Reflection occurs.
CHAPTER III

SPACE SUBDIVISION FOR FAST RAY TRACING

The most time consuming operation in Ray Tracing is the computation of intersections of the rays with all the objects in the scene, performed in order to determine which objects are visible from the view point. As indicated by Whitted [2] the intersection computations take up 95% of the total time for generating an image. A good synopsis of the Ray Tracing technique, with a qualitative breakdown of where the calculation time is spent is also given in Glassner[1]. In the Ray Tracing process, intersections comprise an "inner loop" of the algorithm. Each ray must be checked against each object in the scene so that the number of intersection calculations is linear with respect to the product of the number of rays traced and the number of objects in the entire picture. Doubling the number of objects in a scene approximately doubles the rendering time; doubling both the objects and the rays takes four times longer to render the image.

These intersection operations can require any amount of floating point operations - from just a few to many thousands. To reduce the time spent on these intersection calculations one can either implement the intersection process in hardware, or reduce the number of ray object intersections that must be made to fully trace a given ray.

Ray tracing is computationally a very intensive process. The number of basic ray tracing operations $N_o$ for the standard algorithm is a product of the total number of rays $R$, and the number of objects $O$, in the scene to be rendered. The total number of rays (i.e. number of nodes in the ray tracing tree) depends in turn on the rendered
image resolution $I_r$ (number of pixels), the number of light sources $L_s$, and reflective/refractive properties of objects. On an average at any tree level the new reflective/refractive rays are produced by a factor of $q$, and the ray tracing tree has depth $d$, then

$$R_i = I_r \left( \sum_{i=0}^{d-1} q^i \right) (L_s + 1)$$

Assuming that image resolution is 1000 x 1000, the scene consists of 1000 objects (scenes 50,000 objects have been rendered), there are two light sources, and $q$ equals to 1.1, i.e. 10% of intersections, on the average, produce reflection/refraction rays, then the total number of basic ray tracing operations is

$$N_o = Q_s R_i = 1.8 \times 10^{10}$$

The computational cost of each intersection operation depends on the complexity of the objects. While a straightforward algorithm for intersecting a plane with rays requires two dot products, one subtract, and one divide only, the number of floating point operations required to intersect a ray with a full three-degree bicubic parametric patch is as high as 6135. Usually a number of basic patches/polygons is necessary to represent one object.

**BOUNDING VOLUMES**

We can enclose the given 3-D object space into a Bounding volume. A limit can be initialized for the maximum number of objects to be enclosed within this bounding volume. If the limit has been exceeded, the Bounding Volume is split into smaller bounding volumes as shown in Figure 8. The Bounding Volumes are tested to see which objects they enclose.

A ray can be tested in its path to determine which bounding volume is intersected by the ray. If there is an intersection, then we can further test the objects contained in the bounding volume. If the bounding volume itself is not in the path of a
ray, it is saved on the intersection calculations to be done on the objects contained in that bounding box.

![Octree Structure Diagram](image)

**Figure 8.** Bounding Volumes

**OCTREE STRUCTURES**

An excellent scheme for breaking up a space into bounding boxes is by using the Octree technique. The Octree technique is an extension of the Quadtree Technique used for two dimensional specifications. The Octree technique is described extensively in Hackins, Tanimoto and Meagher[13]. An Octree structure allows dynamical subdivision of space into cubes of decreasing volume until each cube (also called a voxel or volume element) contains less than or equal to the maximum number of objects allowed for each voxel. In solid modelling, octrees are used to define the shapes of objects which are difficult to model with primitive surfaces. In that context, each cell of the tree is either occupied by that object, or it is empty. Each occupied cube may contain some information about color, density, or some other attribute of the object, but the cube itself is considered to be either fully filled by the object or disjoint from it.

One can identify a storage overhead associated with any of the schemes which
form an object space subdivision. In the case of the hierarchy of bounding volumes, the description of each volume must be stored, together with a list of all bounding volumes and/or object primitives that it contains. For the Octree structure, a list of all voxels/object primitives contained within each voxel must be recorded; no information describing the voxel bounding planes need be stored however, since this is encoded implicitly in the identifier. Thus, the database consists of two parts: an octree description and object descriptions.

![Diagram of Voxel, Sphere, Unassociated object, Intersecting object, Enclosing object](image)

**Figure 9.** Object Bounding

The algorithm given assumes that each voxel holds a list of the objects contained in it. The object may be fully contained in the voxel or a part of its surface may be in the voxel as shown in the Figure 9. If the voxel itself is contained completely in the object, one need not assume that this object belongs to that voxel. This is because when a ray is tested against a sphere, only those points on the sphere where the ray pierces the sphere's surface are taken care of.
Figure 10. Octree Hierarchy
BUILDING OCTREES

Octrees can be conveniently used as spatial compartments. They allow dynamic spatial resolution. Volumes with high object complexity can be recursively subdivided into smaller and smaller volumes generating new nodes in the tree for only these new volumes. An Octree structure is shown in Figure 10. This structure of octrees has been coded in 'C' using linked lists. The structure of a node consists of members which include a subdivision flag, object list pointer to the object contained in the voxel, the minimum and maximum extents of the voxel, and the pointers to eight child nodes.

```c
struct OCTREE {
    int flag;
    int <pointer to list of objects in the voxel>
    float <Minimum and Maximum extents of the voxel>
    struct OCTREE *node[8];
}
```

The parent is declared as a pointer to the entire octree structure as:

```c
struct OCTREE *Parent;
```

Memory is dynamically allocated using the memory allocation function in 'C' giving the advantage of memory allocation only when objects are present in the voxel.

```c
Parent = (struct OCTREE *) malloc ( sizeof (struct OCTREE) ) ;
```

If there is no need for the current voxel to be further divided up, the child nodes are declared to point to NULL.

When a ray fails to hit any objects in a given voxel, it must move on to another adjacent node in the path of the ray. Glassner[1] uses a method in which the
parent node is labeled as node 1. When a node is subdivided, it passes its name as a prefix to all of its children, which are numbered from 1 to 8. Thus eight children of the Parent node are named as nodes 11 through 18. The children of node 14 are nodes 141 to 148, and so on.

He uses a hashing scheme, in which the name of a node is hashed into some small number and then follows a linked list of all nodes that hash into that number starting at a given point in the table. By changing the size of the table, any point in the continuum described above is picked. This hashing function returns the node name modulo the table size.

Once a node is established, the node is searched through the appropriate linked list for its entry and associated object list. The number of nodes stored as entries in the hash table/linked list structure is also reduced by a factor of eight. Here, instead of allocating memory to each of the eight child voxels, a block of memory large enough to hold all eight nodes is used. Only the first child is stored in the hash table/linked list structure. The other children are found by adding the right number of node lengths to the first node’s address; i.e., adding one node length for node 2.
A node number containing a point \((x,y,z)\) is found with the scheme as shown below:

```c
findnode(x,y,z) {
    node = 1;
    WHILE ( node subdivided is TRUE )
    {
        IF (x > node_center_x)
            IF (y > node_center_y)
                IF (z > node_center_z)
                    node = (node*10) + 8;
                ELSE node = (node*10) + 4;
            ELSE
                IF (z > node_center_z)
                    node = (node*10) + 6;
                ELSE node = (node*10) + 2;
        ELSE
            IF (y > node_center_y)
                IF (z > node_center_z)
                    node = (node*10) + 7;
                ELSE node = (node*10) + 3;
            ELSE
                IF (z > node_center_z)
                    node = (node*10) + 5;
                ELSE node = (node*10) + 1;
    }
    RETURN ( node )
}
```

The object list pointer points to the head of a list of integers in a dynamically allocated array. The pointers indicate the identity number of the objects contained in the voxel. At the end of the pointer list an illegal object number -1 is used in order to indicate the end of the pointer list.

The list of objects which have surfaces passing through a node are examined. If the total number of objects contained in the list exceeds the limit (in this case 8 objects are considered as the limit per voxel) the voxel is further divided into child voxels. These objects are tested within the child voxels and if any of the child voxel object list exceeds the limit, then the voxel splitting procedure is continued further into that particular child voxel.
To find if an object’s surface is contained in a voxel, the object is intersected with each of the six planes that bound this voxel. If any of the points of intersection lies within the square region of the plane that represents the side of the voxel, the object is considered to be associated with the voxel. Otherwise some point within the object must be examined. If that interior point is within the voxel, the object is with the voxel; otherwise that object is disregarded from the object list of the voxel.

In the case of spheres, the coordinates of the center of the sphere are added with the positive and negative values of its radius in x, y and z directions. Each of these six values is used to find if the surface of the sphere intersects, or is inside the voxel.

In the case of polygons, the polygon vertices are compared with the voxels minimum and maximum extents. If this test fails, it means that either the polygon is completely outside the voxel, or the polygon intersects with the voxel. Hence the Sutherland’s Polygon Clipping Algorithm [17] is used to clip the polygon with the voxel’s 3D volume.

**RAY VOXEL MOVEMENT**

After a ray has finished with a particular voxel, it is important to determine how the next voxel in the direction of the ray is found. The algorithm to get to the next voxel is determined by two important facts. First, since the space is dynamically resolved when the octree is built, there is no knowledge of what the size of the neighboring voxel would be. This can be seen in Figure 11.

Second, this voxel movement operation must be accomplished as fast as possible. This is very important because the time saved by cutting down the ray-object intersections would now be lost for voxel movement operations.
Figure 11. Voxels

Glassner’s [1] idea behind the voxel movement algorithm is to find a point that is guaranteed to be in the next voxel whatever be its size. This point is then used to derive a voxel name (and its associated size and object list) as described in the schemes presented in the previous section.

Figure 12 shows the six planes defining the boundary of a voxel. The intersection of the ray with these six planes is determined using the parametric equation of lines.

Glassner’s[1] algorithm intersects all the six planes. Two of those intersections give the bounds parallel to the X axis, two other give the bounds parallel to the Y axis, and the remaining two intersections give bounds parallel to the Z axis. The algorithm used in this thesis is a modification of the Glassner’s algorithm. Instead of finding intersections with all six planes, certain properties of the ray direction are used, and only three planes are intersected.

For any ray direction, each of the three vector components in X, Y and Z has
either positive or negative values. In a voxel, any ray direction with a positive X will not intersect the left vertical plane of the voxel. Any negative X will also not intersect the right vertical plane. This concept is used for Y and Z components of the direction for the top/bottom and front/back planes of the voxel, respectively. Thus in the proposed algorithm in this thesis, the intersections have been effectively reduced from six to only three planes.

Figure 12. Ray - Voxel Boundary Intersections

It is also inexpensive to intersect a ray with one of these planes, since it costs only one subtraction and one divide operation per plane.
The intersection point derived from the above method may lie on any of the six planes. The point can either be inside the boundaries of a plane or they may intersect an edge shared by two planes, or a corner shared by three planes.

![Diagram of voxel movement](image)

**Figure 13. Voxel Movement**

A point can be found which is guaranteed to be in the next voxel by moving a certain distance from the intersection point on the plane. Since the algorithm has no idea of the size of the next voxel, this distance should surely be a value less than the dimension of the smallest voxel in the entire octree structure. Hence when building the octree, the length of a side of the smallest voxel is preserved. Denoting this distance by Min_len, the algorithm must move from the intersection point on the plane by a
value less than \text{Min\_len} (say \text{Minlen}/2) which ensures that the point is in the next voxel.

As shown in Figure 13, the algorithm must move in one direction if the intersection point is on one of the sides. It moves two directions if the intersection point is on an edge, and in three directions if the intersection point is on a corner.

The function \text{findnode}(x,y,z) in the algorithm is now used to get the node number of the voxel which houses the guaranteed point. This node number is then used to traverse through the linked list structure to obtain the details corresponding to that node.

This thesis presents a different approach than that of Glassner[1] to find the next voxel. In the algorithm by Glassner[1], for every intersection point in the plane of the voxel, the algorithm moves by a minimum distance to ensure a point in the next voxel, and then it iterates using the function \text{findnode}(x,y,z) in the algorithm to get the number of the node to which this point belongs. This is followed by a traversal through the linked list to get the details of that node. In any typical object environment a ray would have to pass through several such voxels.

Kobayashi, Nakamura and Shigei [3] have implemented the Adaptive Subdivision Graph for their multiprocessor ray tracing. As an octree is hierarchical and the positional relation of each leaf node is kept in the node at the upper level, each processor must have access to the common database in order to obtain the relationship between subspaces. They have proposed an Adaptive Subdivision of an object space in network fashion calling it as the Adaptive Division Graph. In this Adaptive Division Graph, each subspace has positional information regarding its face-neighboring nodes. The whole object space is achieved by connecting the face neighboring subspaces to each other. As this graph is not tree-structured and the positional relation of each subspace is geometrically kept, the face-neighboring subspaces on the same level of the
The octree hierarchy are directly connected. Here, a level means the size of a subspace. When face-neighboring nodes are on a lower level, they are connected via a quadtree in view of the interface between one surface of the current level subspace and the surfaces of lower level subspaces. In order to generate the graph, the object space is adaptively divided by using the octree method. This tree is then transformed into the Adaptive Division Graph structure. The input to the algorithm is an object space represented by the octree structure and the output is an Adaptive Division Graph.

The transformation from the octree to the graph is achieved by finding the face neighboring nodes(subspace) for each leaf node (subspace) and connecting them. On each axis the relationships between octants are represented as:

The octant number of nodes faced in the x direction is $a'_x a'_y a'_z = A'_x a_y a_z$ ...... \[42a\]
The octant number of nodes faced in the y direction is $a'_x a'_y a'_z = a_x A'_y a_z$ ...... \[42b\]
The octant number of nodes faced in the z direction is $a'_x a'_y a'_z = a_x a_y A'_z$ ...... \[42c\]

Here $a_x a_y a_z$ is a binary representation of the octant number, where $a_i$ is called the $i$ direction bit. $A'_i$ is $a_i$'s complement. The direction bit of a node is 1 in order to find the node in the minus (-) direction, and 0 in order to find the node in the plus (+) direction. If the node does not satisfy this condition, there is no destination node or prospective node in the sibling nodes, and then parent node of this node is to be examined.

For the leaf node of an octree, the face neighboring nodes in +X, -X, +Y, -Y, +Z and -Z directions are connected. In order to locate the node in -X direction, if the octant is encoded as $a_x a_y a_z$ and $a_x = 1$, the destination node is the node whose octant number is $a_x a_y a_z$ in the brother nodes. If $a_x$ is 0 the tree is ascended until $a_x$ is 1. If this cannot be found then the destination node is out of space. The octant number of this visited node is pushed into a stack. If the destination node is not a leaf node the octant number is fetched from the stack and the x direction bit is changed into its
complement. The tree is descended according to this modified octant number. This process is continued until the stack is empty or the node is the leaf node.

If the leaf node is reached before the stack is empty, this node is the destination node of the specified node and is directly connected to the specified node. If the leaf node is not reached before the stack is empty, or if the destination node is not a leaf node in the first step, the destination nodes are on a lower level than the specified node and some leaf nodes abutting a face of the specified node are connected via a quadtree as the destination nodes. The quadtree of the leaf nodes faced in a specific direction is the subtree of an octree and is generated by picking up the nodes which have a specific octant number. For example in the case of nodes facing -X direction of the specified node, the subtree in which the X direction bit of the nodes is 1 is chosen, i.e., the octant numbers 100, 101, 110 or 111 is chosen. The nodes in the +X, -X, +Y, -Y, +Z and -Z directions are found by means of a similar process.

**Complexity of the above algorithm**

The algorithm mentioned above takes the maximum execution time to transform a complete octree into an Adaptive Division Graph. If $8^k$ represents the number of leaf nodes where $k$ is the number of levels in the octree hierarchy, the algorithm visits one leaf node and connects face neighboring nodes. The face neighboring nodes in three directions are found by two pointer references, since they are brother nodes. For the face neighboring nodes in the other three directions, $2k$ pointer references are required, since the algorithm ascends the tree $k$ levels and descends the tree $k$ levels at the worst. The cost of traversing an octree to visit the leaf nodes is proportional to the number of nodes in the octree. If $n (= 8^k)$ represents the number of leaf nodes, the cost of the algorithm is given as:

$$\left(2n \log n + 6n + (8n - 1) / 7\right) C_p$$

where $C_p$ is the cost of the pointer reference. Thus, the complexity of this algorithm is $O(n \log n)$.​
NEW ALGORITHM

When a ray intersects a side of the voxel, the probability that the intersection point is within the boundaries of this side is far greater than the probability that the intersection point may lie on either an edge or a corner of the side. This is obvious when one compares the area covered by all sides of a voxel to the area covered by its edges and corners. This argument brings to a point that the next voxel intersected by the ray is more probable to share a side with the present voxel, rather than sharing only an edge or a corner.

TABLE I

<table>
<thead>
<tr>
<th>Neighbouring Node Relationships: Map I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
</tr>
<tr>
<td>N(0)</td>
</tr>
<tr>
<td>N(1)</td>
</tr>
<tr>
<td>N(2)</td>
</tr>
<tr>
<td>N(3)</td>
</tr>
<tr>
<td>N(4)</td>
</tr>
<tr>
<td>N(5)</td>
</tr>
<tr>
<td>N(6)</td>
</tr>
<tr>
<td>N(7)</td>
</tr>
</tbody>
</table>

where N(0) to N(7) represent the child nodes 0 to 7, and P(L), P(R), P(U), P(D), P(F) and P(B) represent the left, right, up, down, front and back faces respectively of the Parent node.

In any octree structure, when a node is split into smaller nodes (8 child nodes as in our octree), we find a fine correspondence between the 8 children. As shown in Figure 8, node 0 has nodes 1, 2 and 4 as its immediate neighbors on three sides. It
shares three neighbors of its Parents node on the other three sides. For example, at the top of the octree hierarchy, the Parent node is considered to be surrounded by NULL volume. Hence the child node 1 has nodes 0, 3 and 5 as its three neighboring nodes, and the other three neighbors are NULL. Every node can be then shown to have 6 nodes as their immediate neighboring nodes. This relationship is shown in Table I for all 8 nodes.

At any level in the octree hierarchy, the 'node to node' neighbour correspondence does not change. Every node has six pointers pointing to six neighboring nodes. Thus the structure of the octree node introduced in this thesis has an enhancement over the previous structure proposed by Glassner[1]. It also has the advantage over Kobayashi's[3] algorithm for connecting face neighboring voxels. In Kobayashi's[3] algorithm the face neighboring relationship can be established only after the entire octree structure of the object space is established. The input to their algorithm for generating the Adaptive Subdivision Graph is an object space represented by the octree structure. Each leaf node in the octree is visited to establish the face neighboring node relationship.

The algorithm presented in this thesis uses the natural phenomenon of 'message passing'. As observed in real life traits are passed down generations; from parents to their children or from a society of one generation to the society of the next generation. This principle is implemented in the proposed algorithm to establish the neighboring node relationship. Information regarding neighboring voxels is passed to the child nodes from its parents. The child nodes representing as children of one family establish relationship between themselves and in turn pass their relationship to their respective children. This algorithm has the excellent advantage of establishing face neighboring relationships between nodes while the octree structure of the object space is being generated. Whereas in Kobayashi[3], the octree has to be created first and then is
involved in a complicated method which visits each leaf node of the octree to establish the face neighboring relationship. The structure of the OCTREE used in the new algorithm is as follows:

```c
struct OCTREE {
    int flag;
    int < object list pointer >
    float < Minimum and Maximum extents of the voxel>
    struct OCTREE *node[8];
    struct OCTREE *surround[6]; /* NOTE SURROUND POINTERS */
};
```

The above algorithm creates a massively threaded octree structure as shown in Figure 14. This new algorithm enhances the finding of a node number given a point within the voxel. The algorithm implemented here does not number the nodes of an octree as done by Glassner[1] and Kobayashi[3]. Thus, instead of returning the node number corresponding to the child node, the pointer to the child node itself is returned. Thus we directly traverse through the linked list instead of having to find the node number first.
Figure 14. Massively Threaded Octree Structure
findnode(parent_node, point) {

    IF ( parent_node_has_children is TRUE )
    {
        IF ( point.x > parent_node_center_x )
            IF ( point.y > parent_node_center_y )
                IF ( point.z > parent_node_center_z )
                    child_node = parent_node->child[7]
                ELSE
                    child_node = parent_node->child[3]
            ELSE
                IF ( point.z > parent_node_center_z )
                    child_node = parent_node->child[5]
                ELSE
                    child_node = parent_node->child[1]
        ELSE
            IF ( point.y > parent_node_center_y )
                IF ( point.z > parent_node_center_z )
                    child_node = parent_node->child[6]
                ELSE
                    child_node = parent_node->child[2]
            ELSE
                IF ( point.z > parent_node_center_z )
                    child_node = parent_node->child[4]
                ELSE
                    child_node = parent_node->child[0]
    }

    RETURN ( child_node )
}
Figure 15 represents two pointer mappings used in the proposed algorithm. Map 2 represents the mapping of child nodes of one voxel with the child nodes of the neighbouring voxel. This mapping is useful in order to point to neighbouring nodes of the same size, whose parent voxels are not brother nodes and are located higher up in the octree hierarchy. Map 3 is a pointer mapping used for pointing to nodes which share either an edge or a corner of the node. These two pointer mappings reduces the total number of calls to the Findnode( ) function module. The above mappings have shown considerable speed up of the Ray Tracer, improving tracing speeds by about 20%.

In the proposed algorithm we do not number the nodes corresponding to its parent. In order to find the next voxel, we just need to find the intersection point on the voxel, and determine the side of the voxel the intersection point is in. A pointer corresponding to that side is obtained by the previously established neighbour voxel pointers, called Surround Pointers in this proposed algorithm. Thus several iterations of the function findnode(x,y,z) in the algorithm are saved as well as the time saved on traversing the linked list structure.

In the algorithm proposed by Glassner[1], each time the side of the voxel corresponding to the intersection point is found one has to obtain the voxel by traversing from the top of the octree structure. In a typical object environment consisting of thousands of objects, the depth of the octree would be very large and the rendering time which was destined to be saved by building the octree bounding on the objects, will be lost in just traversing the octree hierarchical structure to find neighboring voxels. The neighboring voxel pointer references implemented in this thesis and the one presented in Kobayashi[3] are very efficient in locating the neighboring voxel.
Map 2

Map 3

Figure 15. Pointer Mappings
Note that the proposed algorithm uses 6 pointers, each capable of pointing to 6 immediate neighboring nodes. There is always a good probability that some of these neighboring nodes would have been divided further. For such neighboring nodes we still have an advantage over the algorithm presented by Glassner[1]. The algorithm need not have to look for the node from the top of the octree hierarchy everytime. Instead, it directly starts to look for the node, considering the neighboring node as the starting point in the child node search. In Kobayashi[3] the leaf nodes abutting the face of the specified node are connected via a quadtree as the destination nodes. The quadtree of the leaf nodes faced in a specific direction is the subtree of an octree, and is generated by picking up the nodes which have a specific octant number. In their cases, since each node is numbered with an octant number, if the X direction bit of the nodes abutting the face is 1, then nodes having the octant numbers 100, 101, 110 and 111 are connected via a quadtree.

There is one major problem involving this kind of implementation. Consider two neighboring voxels A and B. Let voxel A be completely empty or sufficiently full with objects so that no further subdivision is required, and the voxel B contain an extremely large number of objects. By natural octree subdivision the second voxel space will be further subdivided until all objects are minimally bounded according to the constraints set up at the beginning of the algorithm. It is possible that the side of voxel A adjoining voxel B would be sharing several voxel faces of the subdivided voxel B. It is impossible to determine this number during initialization. Hence, one cannot predict the number of pointers which would be required by voxel A to point to the subdivided voxels on its face neighboring voxel B. Moreover, even if this number of pointers is established, one would have to determine which subdivided voxel in voxel B adjoining the face of voxel A is penetrated by the ray traversing from voxel A towards voxel B. In the worst case, this would lead to the inside/outside test of the ray intersection point with each of the boundaries of the face of subdivided voxels
adjoining voxel A, which is a very inefficient method.

In the case when a ray intersects an edge or a corner, the algorithm proposed in this thesis uses the pointer mapping as shown in Map 3 of Figure 15. Results presented in Chapter 5, have shown that the above new algorithm considerably reduces the image rendering time. This problem is not supported in the algorithm proposed by Kobayashi[3]. In that the ray is considered to traverse only between the face neighboring voxels. If a ray intersects a corner or a side of the voxel, Kobayashi's algorithm makes unnecessary ray-object intersection calculations with a neighboring voxel as connected by their algorithm described earlier. The ray would then traverse to the next voxel as determined by its direction and their algorithm continues its ray-object intersection calculation. In the worst case these unnecessary ray-object calculations would be done for two or three voxels before reaching the real voxel.

Thus in the method presented in this thesis we have a more efficient method of constructing the face neighboring relationship between nodes. There is no need for the algorithm to visit every leaf node in order to determine its face neighboring nodes as presented by Kobayashi[2], and there is also no need for the construction of quadtrees for the face abutting nodes. Also the construction of this face neighboring relationship is done while the octree is being constructed on the space subdivision. But in the case of Kobayashi, the octree structure has to be constructed and fed as the input into their adaptive division graph algorithm in order to find the face neighboring relationship. Finding the face neighboring relationship is a very important part of the Ray Tracing algorithm. The time saved in reducing the number of ray-object intersections by subdividing the object space, should not be spent on finding the face neighboring nodes. For any given scene, both the algorithms presented in this thesis and the one of Kobayashi[2], outpace the method of node finding by Glassner[1].
CHAPTER IV
PARALLEL PROCESSING

The high computational requirements of typical graphics algorithms makes the possibility of achieving higher performance through parallel execution very attractive. Many algorithms contain inherent parallelism [26], but taking advantage of this parallelism can be very difficult. One can define three optimizations to improve parallel processing performance.

* Minimizing Communication Overhead

The interprocessor communication time of a parallel computer is typically the major determinant of the difficulty in writing parallel programs for it. Communication overhead can be reduced by batching small messages into a smaller number of large messages, or by recomputing a required value rather than fetching it from another processor.

* Minimizing Synchronization

Trying to synchronize two processes will end up in the processes waiting for each other. This synchronization is caused by functional dependency, where one process needs the results of a remote computation before it can proceed. In a parallel program, synchronization becomes an added cost. Hence parallel programs running on a single processor will often run slower than an equivalent sequential program.

* Even distribution of work among processors

Dividing a problem into a number of parts may not solve the problem. This is because if one part does a larger percentage of the total work, the program may
run very slow due to the communication and synchronization overheads.

To satisfy the above three optimizations an effective program decomposition is needed. Programs can be decomposed efficiently by either Data Decomposition or Functional Decomposition or a combination of both [32], [26].

The Data Decomposition involves dividing the data to be computed, and leaving the same procedure running on each processor. In graphics this is typically done using the image space decomposition, where each processor can be allocated a separate part of the output image. In the object space decomposition each processor is assigned different objects to render.

The Functional Decomposition involves dividing a program into separate parts and assigning these parts to different processors. In a graphics system, one processor could handle geometric transformations, another one clipping, a third one hidden-surface elimination and so on. This type of decomposition is useful when different processors can have different capabilities.

Each of these methods has its drawbacks. The image decomposition requires less communication between the processors, but work may not be equally allocated among the processors because some parts of the image may contain more objects to be rendered than other parts. Object space decomposition has a high communication overhead, for example as in the case of hidden surface removal. Functional decomposition also requires too much synchronization, resulting in the processor with the most work to do, becoming the bottleneck.

**Tightly Coupled and Loosely Coupled Architectures**

One can basically define two different types of parallel architectures [22]. Tightly bound parallel architectures, also called Shared Memory Systems, share a common address space among all processors. Loosely bound parallel architectures are also called Distributed Memory Systems, where each processor has its individual memory
and the processors communicate by sending messages.

The table below [32] shows some of the Advantages and Disadvantages of both types of architectures. Advantages have been marked below.

<table>
<thead>
<tr>
<th></th>
<th>Shared</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost/Processor</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Ease of Use</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Total Memory Capacity</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Remote Data Access</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Pardonable Abilities</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Expansion Capabilities</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 16. Shared versus Distributed systems

Accessing remote data via simple memory reference, is more familiar to programmers who are generally familiar with the conventional sequential programming. Hence the Shared Memory Systems are considered easier to use than the Message Sending Distributed Systems. However, using conventional programming techniques on the Shared Memory parallel computers can often have disastrous consequences, which is one of the reasons that the parallel computers are considered difficult to program.

The Distributed Systems are much easier to expand, since there are no shared hardware resources to become overloaded. They are also much less expensive than the Shared Memory Systems. Distributed Systems with thousands of processors have been built. Since the Distributed Systems are generally superior to the Shared Memory Systems, there is a need to make the Distributed Systems as easy to program as the Shared Memory Systems.
THE LINDA PARALLEL PROGRAMMING ENVIRONMENT

Linda is a parallel programming language developed at Yale University. The Linda parallel communication primitives create dynamic communication structures making parallelism easier to create and utilize. Processes in Linda communicate via tuple space [32]. Processes place tuples in tuple space and remove other tuples using three primitives: 'out', 'in' and 'rd'.

For example the call out("james",23) places a tuple in the tuple space. Now the space contains two elements, the string "james" and the number 23. This tuple could then be removed from tuple space using: in("james",?i), which assigns the value 23 to the variable i. A parameter beginning with a question mark (?) is a formal parameter, the remaining parameters are actual parameters. An actual parameter must match its corresponding tuple field precisely. When a matching tuple is found, values are assigned to the formal parameters, if any, and the tuple is removed from the tuple space. The rd primitive is like the in primitive, except that it does not remove the tuple from the tuple space, but only makes a copy of the tuple.

If an in or rd primitive could match more than one tuple, one of them is chosen arbitrarily. If there are no matching tuples, the in or rd primitive will block until one is inserted into tuple space. The out primitive never blocks.

Sending messages.

Different styles of communication can be implemented using the Linda Communication Primitives. A Message Send can be implemented using the 'in' and 'out' primitives. This is shown in figure 17.

The advantage of such a Message Send is that the sender does not need to know the receiver, and the receiver need not know the identity of the sender. Messages are identified by their contents and not by the receiver or the sender.
The `out` and `rd` primitives can be used to broadcast a single message to multiple receivers, as shown in Figure 17.

**Figure 17. Linda Messages**

**MEMORY CONTENTION**

Memory updating is one of the major problems in parallel processing when several processes need to update a global variable, especially in a shared memory system. On a distributed message sending system, the global variable can be wrapped with a process and programmed to protect the variable from simultaneous update. On a shared memory system, a monitor or a critical region is similarly used.

Linda offers a much simpler solution. The following example demonstrates the
effective use of Linda Primitives for updating shared memory.

\[
in("x",?i) \\
out("x",i+1)
\]

The in primitive removes its matching tuple from the tuple space. If another process tries to increment the value of x at the same time, it will block until the tuple is put back by the out primitive.

**PROCESSOR ALLOCATION**

A decomposed program must be effectively assigned to processors [32]. In static allocation, each piece of work is assigned statically to a specific processor, before the program is run. Load balancing can also be used, where work is made to migrate at run time, from overloaded processors to less busy processors. Communication paths between the different parts must be established. The user may have to explicitly specify the communication paths when the program is written. Greater processor utilization is derived by dynamic allocation of processors and communication paths.

The problem of static allocation can be overcome by implementing a Processor Farm. For example, instead of statistically assigning different parts of the screen to different processors, the screen is divided into many more parts than there are processors, and each processor, called a worker, is assigned work as it needs it. In Figure 18, a single task creator process divides up the screen into chunks of 32 pixels per square. Each worker uses the in primitive to get a description of the area of the screen it is to render. When the worker is done, it out's the results and then in's a new task description. Here the task creator does not know the identity of the workers and vice versa.

The above idea is used in this thesis to parallelize the ray tracer. It is very clear that the raytracing algorithm repeats itself for every pixel in the resolution defined. We can parallelize the ray tracer so that each processor works on a single pixel, or a scan-
line, or a block of pixels from the screen.

A typical screen resolution defined for raytracing is of the order of 512 * 512. If one wants each processor to work on a single pixel at a time, then 512 x 512 tuples would have to be broadcasted into the tuple space. This is not feasible due to memory requirements. This can be modified, and tuples can be dynamically broadcasted depending on the worker's processor consumption. There is however a greater inefficiency here, since broadcasting and receiving by the worker processors can take longer time. This inefficiency was tested, and it was found that in some cases this kind of broadcasting was even slower than working on the entire resolution using sequential processing of pixels on a uniprocessor.

In this parallel ray tracer parallelization is implemented on the "Scanline by
Scanline" principle. For a 512*512 image, the Master process needs to broadcast only 512 tuples. The workers receive these tuples and work on each scanline. The worker processors read the scanline tuples from the tuple space one by one as they are done with computation of the current scanline. Thus Load Balancing is achieved as worker processors are busy working on scanlines as long as there are scanlines present in the tuple space.

THE COGENT XTM PARALLEL MACHINE

The XTM workstation [33] is housed in a 6 x 15 x 15 inch cabinet, attached to an external display, keyboard and mouse. The cabinet contains dual T800 processors connected with 4 transputer links. Each processor has 4MB of memory and is capable of 5MIPS and 1.5MFLOPS. The cabinet also contains input/output devices including serial ports, 90MB or 190MB Winchester disk, and 800KB floppy disk. The images in this thesis were developed on a XTM Computer with a Video Card (1024 x 808) of 8 bits/pixel (grey/color). Very high quality images can be rendered using an alternative Video Card (1024 x 768) with 24 bits/pixel (color).

Extra processors for a parallel processing job can be booted from the resource server. The Resource Server module contains a 16-slot backplane in a 6 x 15 x 18 inch cabinet. The slots are connected with a high performance 32-bit communication bus and four crossbar switches. The Resource Server may be populated with any combination of XTM Compute Cards, with two processors each, and XTM Fiber-Optic communication cards.

KERNEL LINDA

The XTM implements an alternative architecture and implementation for Linda. The system is called Kernel Linda. It is designed to support system-level communication (like the QIX operating system and PIX user interface). It is a run-time kernel
upon which other versions of Linda are built. It provides extensions beyond those in traditional Linda implementations, including multiple tuple spaces, language-independent interprocess communication, and a shared object space.

Kernel Linda is not oriented toward any particular hardware or system; it can be implemented efficiently on a variety of parallel systems (or even sequential machines). In addition, Kernel Linda is able to take advantage of specialized hardware, including the Linda bus and fast link switches of the Cogent Research XTM workstation.

Kernel Linda is not oriented toward any particular language or language class. It provides a layer underneath an actual Linda implementation. Kernel Linda can be used to efficiently support any of the Linda systems. A compiled language such as C or Fortran can use a preprocessor to translate Yale Linda calls into Kernel Linda, an interpreted language such as Prolog or PostScript can implement its Linda calls in terms of the Kernel Linda primitives, and system tools such as a debugger can use the Kernel Linda primitives directly. In addition, Linda calls in one language can access tuples created by a program in another language.

KERNEL LINDA DATA TYPES

Kernel Linda supports a set of language independent data types. They support Linda operations and provide shared object space (a controlled form of shared memory), which can be implemented directly on a system that contains shared memory, or simulated efficiently on a distributed system (such as the Cogent Research XTM).

There are two types of data types in the Kernel Linda.

Simple Types (such as names, integers, reals, and values) are always passed by value. Using the 'out' function described earlier, a Linda object is placed into tuple space, and the object is copied into the memory owned by the Kernel
Linda.

Composite Types (dictionaries, arrays, strings and blocks) are always passed by reference, using a locator. A locator is an indirection to an object, which allows a reference to point to an object on another processor, and also makes memory management possible.

The built in types are:

* dictionary - A set of Key / Value pairs.
* array - An array of Kernel Linda objects.
* string - A string of byte-sized characters.
* block - An uninterrupted hunk of raw memory.
* name - Identifiers used as keys in dictionaries.
* int - A 32 bits long word-sized integer.
* real - A 32 bits long word-sized floating point number.
* real64 - A double-precision floating-point number.
* struct - A hunk of user supplied data, passed by a value.
* null - indicates error or null object.

The first four types are called Composite Types, since they have internal structure. The remaining six are Simple Types. Simple types are always passed by value, and composite types are always passed by reference using a locator.

Objects have flags associated with them.

* executable - Indicates that the object is executable. An executable block is a load module, compiled from a language such as 'C', 'C++' or Fortran. An executable string is a shell script.
* readonly - A readonly composite object can only be read and not modified.
Appendix A contains two program codes implementing Kernel Linda binding on 'C'. The programs are the Master process and the Worker process, and represent most of the data types mentioned above.

Linda can be used to implement many different styles of parallel programming. It can be used like a shared memory system, but without the associated dangers. It can be used like a distributed message-sending system, but with much more dynamic communication. Linda can be used to implement parallel programs based on data or functional decomposition, and its flexibility allows the decomposition of a problem to be easily tuned for maximum performance.

Many problems have inherently parallel solutions; that is why pseudoparallelism (multiprogramming) is so common. Examples of this are the interactive graphical user interfaces such as NeWS from Sun Microsystems, the multitasking operating systems such as UNIX, and the programs to solve graph problems such as 8-queens. Since the pseudoparallel programs already pay the cost of synchronization, they sometimes show superlinear speedup when run in parallel (n processors run more than n times faster than one processor).

PARALLELIZATION OF THE RAY TRACING ALGORITHM

There have been several methods proposed for the parallelization of the ray tracing algorithm. Several architectures for hardware implementation of the Ray Tracing Algorithm have been proposed. Recent progress in VLSI technology has made it possible to achieve large scale parallel processing. For fast image synthesis, parallel processing is applicable to calculate the pixel values on a screen since the intensity of each pixel can be independently calculated. Such a parallel processing approach is also called as Pixel-Oriented parallel processing. Since Ray Tracing is an object space algorithm in which hidden-surface removal is carried out in an object space, each processor requires object description regarding the whole object space. Therefore, a tightly
coupled multiprocessor system is required. But as the number of processors in this system increases, so does the intercommunication between the processors and the common memory for object description.

Kobayashi, Nakamura and Sigei[3] present a method in which the object space for image synthesis using Ray Tracing is processed in parallel. An object space is divided into parts(subspaces), each of which is allocated to a processor. Each processor detects simultaneously the intersections of the surfaces of each object and a fixed number of rays over the whole space, and calculates the local intensity on an object in each subspace. The propagation of a ray is realized by the interprocessor communication. The global intensities of pixels on a screen are calculated by the other kind of processors simultaneously. An Adaptive Division Algorithm, as explained in chapter three, for which parallel processing of the object space was implemented.

In the system of Kobayashi [3], the Ray-Tracing algorithm is divided into two processes. The first finds the intersecting object and calculates the local intensity on the object. The second calculates the global intensity for each pixel. The authors show a hardware approach to achieve the objective of separating the complex intersection phase from the shading phase. In order to find the intersecting object efficiently, the relationship between the position of an object and the direction of a ray is taken into consideration. Since the traditional pixel/screen oriented parallel processing approach for image synthesis, as implemented in image space algorithms, is not suitable for finding the intersecting object in Ray Tracing, an object-oriented parallel processing for Ray-Object intersection calculations was proposed.

The object space is divided into subspaces according to the position of each object, and a subspace is allocated to a processor, keeping the spatial coherence. In a subspace, each processor calculates local intensity exerted on the object by the visiting rays. A ray selects the processor with the subspace according to its direction. The pro-
pagation of the ray is achieved by interprocessor communication. In order to calculate the global intensity, a screen oriented parallel processing is used. After determining the intersecting objects associated with each pixel, the calculation of the global intensity for each pixel can be carried out simultaneously. Thus, in their system, two kinds of parallel processing, object oriented and screen oriented are introduced.

In the architecture of the parallel processing system [3] there are five basic parts. A host computer, intersection processors (IPs), shading processors (SPs), an interconnection network, and a distribution network. The host computer allocates the subspaces of an object space and the initial rays to the IPs. The IPs determine whether or not the visiting rays intersect a given object within the subspace and, if so, calculate the local intensity on the object and generate the result packet. Each packet is an information parcel consisting of information about the local intensity and the information necessary to construct the binary shade tree. This packet is transferred to the destination SP via the distribution network. If the ray does not intersect the object, or if the ray is newly generated after the reflection/transmission process, the IP transfers the ray packet to the next IP of one subspace, abutting a face of the subspace (i.e., one of the face-neighbor subspaces) according to the direction of the ray. For shadow calculations the IP generates the ray packet for shadowing and issues this packet in the direction of the light source.

The SPs serve to calculate the global intensity of each pixel simultaneously. The result packet transferred from the IP is stored in the input queue of the SP. In the matching unit in the SP, the binary shade tree of the specified pixel is constructed from the packets generated on the surfaces of each object. Since the growing process of each node of the tree is arbitrary, calculation of the global intensity is carried out in the data-driven manner. Thus, the subtree whose required input becomes available is processed first. The inputs of this subtree are transferred to a functional unit, and the
calculation of the global intensity at this subtree is carried out. The interconnection network connecting the IPs to each other takes care of the local communication between the subspaces facing each other. A Hypercube (binary n-cube) network was used as the interconnection network. In this Hypercube network, a communication link between two processors exists if and only if the binary representation of a processor number differs by only one bit. In the six dimensional Hypercube network where the number of processors is $2^6$ each processor has six communication links. The Hypercube network was used by Kobayashi[3] since the routing algorithm is simple, and an optimal mapping of the adaptively divided object space on to this network could be easily achieved. Communication between face-neighboring subspaces whose sizes are equal were attained using one link. Communication between the face-neighboring subspaces whose sizes are not equal was achieved by using several links according to the difference in their sizes. The distribution network connects the IPs with the SPs. Since the communication between them is not local, Kobayashi[3] used a multistage switching network like an Omega network.

Following the above paper, Kobayashi[4] presented load balancing strategies for a parallel ray tracing system based on constant subdivision of the object space. An object space was divided into regular cubes (subspaces) and allocated to the processors in the system. Load balancing in such a system is an important issue since the constant subdivision would not allocate objects uniformly in the subspaces. Kobayashi[4] proposed both the Static and Dynamic load balancing strategies for such a system. In the category of Static load balancing, the strategies for mapping the subspaces into the processors were evaluated by simulation. In order to realize dynamic load balancing a hierarchical multiprocessor system has been proposed to overcome the limitations of static load balancing in a large scale multiprocessor system. Thus their hierarchical architecture employs the Dynamic load balancing mechanism in addition to the Static load balancing.
The object space is divided into regular subspaces of appropriate sizes. These subspaces are allocated to the respective processors. For interconnection network of the processors, the nearest-neighbour connection network of one, two and three dimensions is used for one, two and three dimensional parallel processing. When allocating regularly subdivided object space to these systems, the problems to be considered are:

1. The optimum number of subspaces of an object space depends on both the number of objects and the position of the objects to be rendered. However the number of processors in the system is fixed. In general, the number of subspaces is larger than the number of processors.

2. In image synthesis, the computational load which is necessary in each subspace is not uniform, since it depends on whether the subspace includes objects or not, as well as on the position of a subspace in the object space.

The first drawback was overcome by allocating a set of subspaces to a single processor. The simplest method implemented called the Block Allocation, was to allocate a block of neighboring subspaces to a single processor. In order to overcome the second drawback the neighboring processors can be made to exchange their computational load at the execution time, according to the load balance between them. This method is however, effective only to a certain degree. Further more, this method is more expensive since controlling the load balance involves the communications between the processors and the movement of the object descriptions between the neighboring processors. For this problem Kobayashi[4] presents a method in which the load concentration is distributed among several neighboring subspaces. In this Distributed Allocation, the subspaces are allocated at certain intervals to one processor, resulting in the neighboring subspaces of a local space to be allocated to different processors.

In the results of their simulations, it was found that the processing time decreases as the number of processors increases. The processing time using the Distri-
Distributed Allocation was two to three times faster than that using the Block Allocation. Thus the Distributed Allocation algorithm was found to be excellent regarding static load balancing. However, the decreasing rate of processing time in this case was small when the number of processors was large. This is because of the relationship between the number of subspaces in an object space and the number of processors in the processor space of the system. When the number of subspaces is fewer compared to the number of processors, the effectiveness of load balancing by the Distributed Allocation can not be obtained since a little local information of an object space is assigned to one processor and the load is not uniformly allocated to processors. Thus, there is the limitation of static load balancing by the Distributed Allocation when the degree of space subdivision is not sufficient, compared to the number of processors.

A hierarchical multiprocessor system with static and dynamic load balancing mechanism was introduced since effective utilization decreased as the number of processors increased in a large scale multiprocessor. A new system architecture was proposed to support Dynamic load balancing with Static load balancing. There are two levels in this architecture: a cluster level and a processing element level. At the cluster level, the subspaces are allocated to each cluster by using the distributed allocation. Thus, at this level the clusters realize an object space to be traversed by rays. At the processing element level, processes allocated to a cluster are carried out simultaneously by the processing elements. Therefore, at the cluster level, the static load balancing is achieved by the distributed allocation, while at the processing element level, the dynamic load balancing is achieved within a cluster in execution time. The system consists of a host computer, k clusters and an intercluster connection network. The host computer controls the system. The intercluster connection network interconnects the clusters, and is used to transfer rays to the appropriate clusters according to the directions of the rays. An N-dimensional nearest neighbour interconnection network is used for this network. The cluster consists of a cluster controller, m processing
elements and a intracluster connection network. The cluster controller receives rays visiting the cluster and assigns them to the processing element which has the lightest load in the cluster. The processing elements carry out the ray object intersection calculations and/or local intensity calculations of intersecting objects for the assigned rays. Communications between the cluster controller and the processing elements are achieved over the intracluster connection network.

In order to examine the effect of static and dynamic load balancing in the hierarchical multiprocessor system, performance of the system was evaluated by simulation. The processing time was recorded as a function of the number of processors in a cluster. The processing time linearly decreased as the number of processors increased, i.e., the speedup was almost linear. The configuration of the hierarchical system consisting of 256 processing elements with one cluster composed of 16 processing elements was used. This configuration revealed a higher effective utilization (84% to 86%) compared to the non-hierarchical system consisting of 256 processing elements whose effective utilization was 20% to 40%. In this case the processing time in the hierarchical system was two to four times shorter than that of the non-hierarchical system and then the hierarchical system can achieve almost linear speedup. Hence a nearly "ideal" load balancing seemed to have been kept in a large scale multiprocessor system by using these static and dynamic load balancing mechanisms.

Green, Paddon and Lewis[8] have proposed a parallel algorithm and Tree-based Computer Architecture for Ray Traced Computer Graphics. They assume that a large number of processors are available, each with a relatively small amount of local memory. Additionally a controller process is given the responsibility of generating the tasks associated with a particular ray tracing application, and allocating these tasks to processors on demand. Their model consists of Work Processors which perform ray tracing operations in parallel, a Distribution Processor which partitions the tasks, and
the results are collected by a Collection Processor which is associated with the display hardware. In general, the problem domain consisted of a very large object database relative to the size of local memory available to each processor. This assumption implied that portions of the database communicate between processors, thus degrading the performance of the system by introducing additional communications overhead.

The processor system is arranged in the form of a tree structure with the controller processor placed at its root. For an $N$ processor system with $k$ subtrees per node, the complexity of the inter-processor communication is $O(\log_k N)$. The advantage of this system over a square array of processors is that the communications are much simplified. When a processor becomes idle, a request is made for work. Since all work originates at the root, this is propagated up the tree. Similarly, if a particular data item is required which is not available locally, it is requested by passing a message up the tree. Tasks and data items are satisfied by passing the relevant message down towards the requesting node. Adopting this mechanism of issuing the requests up and satisfying them down the tree, made communication simple, thereby reducing the overhead of the message processing. To keep this architecture general and to impose as few restrictions as possible a collector network would not keep the control structure simple. Hence, the results of the ray tracing tasks are sent back up the tree to the root, which then forwards them to the display hardware.

The architecture is represented by two processes which run concurrently on the tree structure. First, the root process acts as the controller for the whole network and also as an interface to the host computer and the display hardware. This process is responsible for generating the primary rays to be traced. The second process is the node process. It performs ray tracing operations as they are passed down the tree, and return the completed tasks to the root. The node process receives messages from its parent, and in the case of the non-leaf node process, from its children, and it relays
them as appropriate. It also maintains its own local database of objects which, in general, is much smaller than the description of the database of the whole object space. The node processor maintains two copies of the most frequently used objects. These three independent functions of processors are called the Work Processor, the Work Distribution Manager and the Database Manager.

In this configuration, the secondary rays, due to reflections and refractions are rendered concurrently on a number of processors, since each secondary ray is independent of one another. For each new ray spawned there may or may not be an idle processor ready to receive and begin processing the task. Thus, at each node a ready work stack of tasks is maintained. When a Work Process becomes idle, a request for work is made to the Work Distribution Manager. If the ready work stack is non-empty, then a task is popped and given to the WP to perform. New tasks which are spawned as a result of the current task are passed back to the Work Distribution Manager and pushed on to the stack. In the case where a request for work cannot be satisfied locally, because the ready work stack is empty, a global request for work is issued by the Work Distribution Manager to its parent. Thus, each Work Distribution Manager must monitor requests from both Work Processor and its children.

The bottlenecks in the system are caused by inter-node communications. The use of distributed ready work stacks ensures that requests can be satisfied locally without the need for each request to propagate to the root of the tree. Another feature of the Work Distribution Manager which helps to reduce global communication is the way in which requests are multiplexed. A list of the node's Work Processor and children which require work is kept, and only a single request is passed to the parent node. When a task arrives to satisfy the request, it is given to the local Work Processor. If this is busy then it is sent to one of the child nodes, and further requests for work is made, if necessary. A second data structure is also maintained by the Work
Distribution Manager for task management. This is called the Delay Work List. It is so called because the ray intersection tree cannot be determined until the spawning of subrays from the intersection. Such rays are said to be delayed and placed on the delayed work list until all of the subrays have been rendered.

The parallel algorithm and the ray tracing code were written entirely in Occam, which is a language based upon a model of concurrency where all communication is synchronized and unbuffered, and takes place through the use of a data structure called a channel. Processes communicate by passing messages along channels. The implementation described above requires that a number of distinct messages of varying sizes are to be passed down the channels at different times. The message consists of an array of elements. The first element being a unique identifier which indicates the type of message. Subsequent elements correspond to the data fields for the particular message.

A multiprocessor algorithm for ray tracing is described in Cleary[7]. Two and three dimensional network of processors have been proposed. It was found that a square network of processors performed better than a cubic network for a sufficiently small number of processors. In the cubic network each processor has six links to its six neighboring processors. In the square network each processor has four links to its four neighboring processors. For the square network the 3D scene is divided into rectangular volumes and all the processors are face connected to the frame buffer. In the cubic network the 3D scene implements adaptive division and only the master control processor is connected to the frame buffer.

One of the major disadvantages of the system from Cleary[7] was, uneven load distribution among processors which resulted in partially idle and busy processors. When the number of processors was large there was a degradation in the performance of the square network. Using a cubic network of processors the efficiency decreased
with the decrease in the number of processors. Speedup was less than linear for more than ten processors. Rays communicated between two processors are stored in a queue until the processor is free to process the ray. These queues get rapidly saturated which contribute to the degradation of the efficiency of the system.

Another parallel ray tracing algorithm has been presented by Priol & Bouatouch[5]. Their algorithm subdivides the scene into 3D regions, the adjacency of which is modelled by a connectivity graph of regions. Since a ray tracing process is associated with each region, this graph becomes a graph of processes, the edges of which represent the communication between processes. This graph of processes is suitably mapped onto a hypercube topology in such a way that the communication cost is minimized. Static load balancing is performed and solutions are brought to the problems of network congestion and termination.

VECTORIZATION OF THE RAY TRACING ALGORITHM

Vectorizing Fundamentals

1. A stream of data elements must be available to perform same calculations.

2. Each individual calculation must be independent of the results of other calculations in the data stream.

Vectors in a Ray Tracer

1. In a Ray Tracer computations involved with each pixel are independent of each other.

2. Rays belonging to different pixels of the image are completely independent.

3. Rays spawned by one pixel are not mutually independent.

Plunkett & Bailey[16] use a vectorized computer architecture to speed up the ray-tracing process. When a ray needs to be fired, that request is placed on a ray queue. When the queue becomes full, the vector processor fires all those rays at once.
The program then goes back and uses the intersection results where they were origi­
nally requested. Significant speed increases result from this retooling of the algorithm.

Thus the vectorized ray tracing algorithm is:

WHILE (there are pixels left to evaluate) {

1. Until the queue is full, add more rays. (Some of these rays originate at the eye
   position and are directed through new pixels.)

2. Calculate the intersections of the rays from a entire queue of rays with each
   surface in the scene, one surface at a time, using vector code.

3. Determine which of the above intersections is the visible surface for each ray.

4. From those rays which hit a visible surface, spawn any further rays necessary
   to model special effects. Add these rays to the queue.

5. Determine the intensity of any pixels that have all their visible surface calcula­
   tions complete.

}

In the above algorithm, memory requirement increases drastically as more rays
are spawned. Intersection distances of the ray from its origin to its intersection point
on the object along with the identity of the object has to be stored. The storage
requirement for each ray would then be 2*N*L, where N is the number of objects and
L is the ray queue.

In their example scenes containing 100 primitives results of one queue intersec­
tion was around 20,000 words. This kind of memory is available only on super com­
puters. Hence in their experiments the CDC CYBER 205 super computer was used.
Speed ups of 1225:1 were achieved compared to a VAX for a image with 100,000
ray-sphere intersections.
IMPLEMENTATION OF PARALLELIZATION IN THIS THESIS

The ray tracing algorithm presented in this thesis was parallelized on a Cogent XTM parallel workstation running a Unix-based operating system. A resource server booted up to nine processors to be used in parallel. Each processor has 4 megabytes of memory and hence it is easier to let each processor read the entire object database. This is very efficient for this architecture since there is absolutely no need for interprocessor communication. Image space parallelism is used and hence communication is needed only to allocate tasks of image space to the processors. The new algorithm for object space subdivision described in chapter 3 and implemented in this thesis creates a virtual object space parallelism within each processor. This is achieved since each processor has the description of the entire object space, and also each processor runs independently the new algorithm with has neighbouring node pointer references.

Image space parallelism can be implemented in three ways.

1. Individual pixels
2. Blocks of pixels
3. Scanlines

Distributing individual pixels to processors cause very high communication times. For example in a image consisting of 1000 x 1000 dimensions $10^6$ tasks have to be spawned. The results from each processor add up to another $10^6$ tasks communication overhead. If antialiasing is implemented in such a system the total number of communications would be $2 \times N \times 10^6$, where N is the number of samples per pixel implementing antialiasing.

The above method of parallelization was implemented on nine processors and it was found that the raytracing algorithm was extremely slow. The high communication overhead drastically degraded the system performance. Scanline parallelization was successfully implemented and its results are shown in Chapter 5.
of pixels involve more complications in coding since the results received from the processor has to be rearranged into scanlines. A near linear speed up was achieved with this arrangement on the Cogent parallel system. But the overall speedup with both scanline and block implementations would be the same. The algorithm was divided into two processes. One is the Master Process and the other the Worker Process. The master process initiates loading the raytracer into the individual processors and places scanlines in the Linda tuple space. The worker processors work on the scanlines available in the tuple space. Results of the scanlines are placed back in the tuple space and inputs more scanlines if they are available in the tuple space. The master process collects these scanline results which are then later rendered on to the screen. Thus the processors are kept uniformly busy all the time.

If the number of processors is greater than the number of scanlines in the tuple space there is a degradation in the utilization of the system processors. In such cases the lengths of the scanlines can be shortened in order to place as many scanline tasks as there are processors. The absolute limitation being scanline sizes equal to one pixel. Images rendered in this thesis were 512 x 512 in size and the maximum number of processors available was 9. It is very cost prohibitive to have as many as 512 processors to be running in parallel. Hence the parallelization method implemented in this thesis is ideal for the Cogent system architecture.
CHAPTER V

RESULTS

In this chapter, some ray traced images are shown. These images have been photographed off the Cogent XTM screen. Image space parallelism has been implemented as explained in chapter 4. The raytracer was also implemented on a Sun 386i and its results are presented.

Figures 19 to 23 show the Ray Traced images which were developed using the algorithm proposed in this thesis. The images have the following characteristics.

IMAGE I has 25 diffuse polygons, 1 reflective sphere, 4 diffuse spheres and one light source.

IMAGE II has 50 diffuse polygons, 2 reflective sphere, 3 diffuse spheres and one light source.

IMAGE III has 120 diffuse spheres and one light source.

IMAGE IV has 50 diffuse polygons, 1 highly reflective polygonal mirror

IMAGE V has 68 diffuse polygons and one light source.
Figure 19. IMAGE I

Figure 20. IMAGE II
The images are colormapped and each pixel has only 8 bits of data.

The rendering time of these images are given in the following tables and graphs. The effective utilization of a multiprocessor system is measured as

$$EU = \frac{Upt \times 100}{Mpt \times Np}$$

where $EU$ is the Effective utilization, $Upt$ is the Uniprocessor processing time, $Mpt$ is the Multiprocessor processing time and $Np$ is the number of processors.
Figure 24. Parallel Processing Speed up graph for figure 19.

TABLE II

<table>
<thead>
<tr>
<th>Processors</th>
<th>New_algorithm</th>
<th>Glassner's</th>
<th>Effective Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>138</td>
<td>148</td>
<td>82.88 %</td>
</tr>
<tr>
<td>7</td>
<td>154</td>
<td>175</td>
<td>84.88 %</td>
</tr>
<tr>
<td>6</td>
<td>175</td>
<td>220</td>
<td>87.88 %</td>
</tr>
<tr>
<td>5</td>
<td>205</td>
<td>250</td>
<td>89.92 %</td>
</tr>
<tr>
<td>4</td>
<td>248</td>
<td>300</td>
<td>92.23 %</td>
</tr>
<tr>
<td>3</td>
<td>324</td>
<td>495</td>
<td>94.13 %</td>
</tr>
<tr>
<td>2</td>
<td>470</td>
<td>610</td>
<td>97.34 %</td>
</tr>
<tr>
<td>1</td>
<td>915</td>
<td>1081</td>
<td>100.00 %</td>
</tr>
</tbody>
</table>
Figure 25. Parallel Processing Speed up graph for figure 21.

TABLE III

<table>
<thead>
<tr>
<th>Processors</th>
<th>New Algorithm</th>
<th>Glassner's</th>
<th>Effective Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>141</td>
<td>152</td>
<td>77.12 %</td>
</tr>
<tr>
<td>7</td>
<td>151</td>
<td>173</td>
<td>82.30 %</td>
</tr>
<tr>
<td>6</td>
<td>170</td>
<td>202</td>
<td>85.29 %</td>
</tr>
<tr>
<td>5</td>
<td>201</td>
<td>242</td>
<td>86.56 %</td>
</tr>
<tr>
<td>4</td>
<td>249</td>
<td>286</td>
<td>87.34 %</td>
</tr>
<tr>
<td>3</td>
<td>310</td>
<td>371</td>
<td>93.54 %</td>
</tr>
<tr>
<td>2</td>
<td>454</td>
<td>562</td>
<td>95.81 %</td>
</tr>
<tr>
<td>1</td>
<td>870</td>
<td>1030</td>
<td>100.00 %</td>
</tr>
</tbody>
</table>
Figure 26. Parallel Processing Speed up graph for figure 23.

TABLE IV

<table>
<thead>
<tr>
<th>Processors</th>
<th>New Algorithm</th>
<th>Glassner's</th>
<th>Effective Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>171</td>
<td>192</td>
<td>77.11 %</td>
</tr>
<tr>
<td>7</td>
<td>195</td>
<td>219</td>
<td>77.28 %</td>
</tr>
<tr>
<td>6</td>
<td>220</td>
<td>248</td>
<td>79.92 %</td>
</tr>
<tr>
<td>5</td>
<td>251</td>
<td>278</td>
<td>84.06 %</td>
</tr>
<tr>
<td>4</td>
<td>301</td>
<td>349</td>
<td>87.62 %</td>
</tr>
<tr>
<td>3</td>
<td>387</td>
<td>501</td>
<td>90.86 %</td>
</tr>
<tr>
<td>2</td>
<td>552</td>
<td>702</td>
<td>95.56 %</td>
</tr>
<tr>
<td>1</td>
<td>1055</td>
<td>1220</td>
<td>100.00 %</td>
</tr>
</tbody>
</table>
Figure 27. Effective Utilization of the Multiprocessor System
Results on Sun386i uniprocessor machine

Image with
100 spheres, 4 rectangular mirrors and 2 light sources
Tree depth = 10  Octree Depth = 7
Image Resolution 512 x 512
Lens Setting 50mm.
Corners and Edges  4703
Number of Sphere Intersections = 1,436,061
Number of Polygon Intersections = 526,179
Total number Object Intersections = 1,962,240 (New Algorithm)
Total number of Shaded Intersections = 26,355
Total number Object Intersections = 167,422 x 10^6 (Without Adaptive division)
Tree Search  -> Map_1 216,925  Map_2 89,203  Map_3 7,692
Execution Time -> Map_1 830 secs  Map_2 795 secs  Map_3 775 secs

TABLE V

<table>
<thead>
<tr>
<th>UNIPROCESSOR RESULT I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun 386i</td>
</tr>
<tr>
<td>Execution time</td>
</tr>
<tr>
<td>Tree Search</td>
</tr>
<tr>
<td>voxel intersection</td>
</tr>
</tbody>
</table>
Image with

500 spheres, 5 rectangular mirrors and 3 light sources

Tree depth = 10
Octree Depth = 6
Image Resolution 512 x 512
Lens Setting 50mm.

Corners and Edges 9146
Number of Sphere Intersections = 8,722,455
Number of Polygon Intersections = 864,796
Total number Object Intersections = 9,587,251 (New Algorithm)
Total number of Shaded Intersections = 465,003
Total number Object Intersections = 1.08 \times 10^{12} (Without Adaptive Division)

Tree Search -> Map_1 5,190,338 Map_2 181,652 Map_3 47,382
Execution Time -> Map_1 127 mts Map_2 104 mts Map_3 93 mts

TABLE VI

<table>
<thead>
<tr>
<th>UNIPROCESSOR RESULT II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun 386i</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>Execution time</td>
</tr>
<tr>
<td>Tree Search</td>
</tr>
<tr>
<td>voxel intersection</td>
</tr>
</tbody>
</table>
CHAPTER VI

CONCLUSIONS

Ray Tracing has firmly established itself as the most effective rendering technique for providing the highest quality of image synthesis. It has proved to be a flexible and easily extensible framework. However, to be useful in interactive applications it must be performed quickly. In this thesis I have presented a new method of adaptive subdivision of the object space. This new algorithm establishes an efficient face neighboring relationship between subspaces in the object space. Unlike an octree data structure, it is unnecessary to ascend and descend the tree when traversing between subspaces, since each subspace of an object space locally has information about the relative position of its face neighboring subspaces. The data structure for this adaptive subdivision is also presented. Subspace traversal by ascending and descending the octree was also implemented to compare the speedup of the new algorithm with the previously proposed algorithm by Glassner[1]. His idea has been used by several researchers in implementing their ray tracers. A significant improvement of 10% to 25% was found on the images rendered shown in chapter 5. A larger speedup is expected when complicated images with a large number of objects are used. Efficiency of creating this subspace relationship over that of Kobayashi[2] has also been presented in chapter 4.

In this system, image space parallelism of the ray tracer was implemented. The parallelism was implemented on a Cogent XTM parallel workstation using Cogent’s Kernel Linda parallel programming. Nine T800 processors each with 4MB of memory was available. Since parallelism in image space was implemented, the database of the
scene was distributed to each of the processors. Processing time from one to nine processors have been registered. From the user interface developed by Cogent using the PIX window environment, processor utilization was found to be at its maximum from the histogram bar graph. A near linear speed up of the raytracer was achieved. The effective utilization of the processors was an average 82% which is similar to the effective utilization of processors by Kobayashi[3]. Object space parallelism was not implemented because of very high communication times involved between processors. The new adaptive space subdivision algorithm was, however, used to implement the virtual object space parallelism in each of the processors.

This new raytracing algorithm can be easily implemented on either a uniprocessor machine or on loosely coupled multiprocessor machines (i.e., processors with local memory). Thus, the algorithm can be implemented on any personal computer or distributed on a network (Distributed Processing) so that each of the computers (processors) works on a part of the image.

FUTURE WORK

The raytracer needs to be integrated with a 3D drawing package for easier definition of objects in a scene. Further research in voxel processing needs to be done, where efficient neighbouring node relationship algorithms can be developed. Photorealism can be obtained by supporting texture maps and volume rendering. Surfaces made of Beta Splines needs to be supported so that the Ray Tracer can support most CAD packages. The present program code needs to be refined in order to implement better data structures and efficient usage of memory.
SELECTED BIBLIOGRAPHY


[18] Siggraph 1988 Course Notes "An Introduction To Ray Tracing"


APPENDIX A

MASTER PROCESS
MASTER PROCESS

/* This Ray Tracing program is built in C and uses Kernel Linda for evoking processors on the Cogent XTM machine to run the Raytracer in parallel. Each processor handles an individual scanline which it obtains from the environment's tuple space */

#include "header.h"

main(int argc, char **argv) {
    Val scanline;
    FILE *fp;
    int i, j, No_of_Processors, ticks;
    int elapsed_time(void); /* Time the entire execution */
    float total_time;

    if(argc != 3) {
        fprintf(stderr, "Usage: %s scenename outfile ", argv[0]);
        exit(1);
    }

    /* Get the image resolution to be scanned by the processors */
    fp = fopen(argv[1], "r");
    fscanf(fp, "%d %d", &X_res, &Y_res);
    fclose(fp);

    printf("Input No_of Processors to work on ");
    fflush(stdout);
    scanf("%d", &No_of_Processors);

    /* Start Timer */
    start_timer();

    /* Return a Val for the local process environment 
    Each process executes in an environment that 
    can be accessed via a dictionary */

    Val root = environment( )["Root"];
    root.out((Val)"DICTIONARY", createdict( ));
    root["DICTIONARY"].out((Val)"MEMORY", createdict( ));

    /* Output scanlines to the dictionary */
    for(i=(Y_res-1); i>=0; i--)
        root["DICTIONARY"].out((Val)"SCANLINE", (Val)i);
/* Create a Linda block of the executable file "raytrace" */
Val workers = createblockfromfile("raytrace");
if( workers == NULL )
{
    printf(" Cannot open executable file ")
    exit(1);
}

/* Make an environment copy and start execution of the worker processors */
for(i=0; i < No_of_Processors; i++)
{
    Val env = environment().ecopy();
    env.out("SCENEFILE", (Val) argv[1]);
    int pid = env.execute(workers,i+1);
    if(pid<0)
    {
        printf("Cannot open process ",i);
        fflush(stdout);
    }
}

fp = fopen(argv[2], "w");
if(fp == NULL)
{
    printf("Cannot open to write image file");
    fflush(stdout);
    exit(1);
}

#ifdef XTM
{
    int magic = 0xA5312070;
    fwrite(&magic, sizeof(int), 1, fp);
} #endif

fwrite(&X_res, sizeof(int), 1, fp);
fwrite(&Y_res, sizeof(int), 1, fp);

/* Allocate Memory for each scanline, where each pixel stores R,G & B values */
int *pixel = (int *) malloc(X_res * 3 * sizeof(int));

/* Read Raytraced scanlines from the dictionary */
for(i=(Y_res-1); i>=0; i--)
{
    root["DICTIONARY"]["MEMORY"].in((Val)i,scanline);
    scanline.gets(0, X_res*3*sizeof(int), (char*)pixel);
for(j=0; j < (X_res*3); j++)
    putc((int)pixel[j], fp);

fclose(fp);

ticks = elapsed_time();
total_time = (float)ticks / 15625.0;

printf(" Total elapsed time : ");
printf(" For %d Processors  %f seconds", No_of_Processors, total_time);
fflush(stdout);
APPENDIX B

WORKER PROCESS
WORKER PROCESS

#include "header.h"

void main( ) {
    Val block, i, scenefile;
    int flag[3], tree;
    float pixel;
    struct PIXEL Pixel_Intensity;
    struct PIXEL tree_search( );
    struct COORDINATES AXIS;
    struct COORDINATES normalize( );
    struct OCTREE *root;
    struct OCTREE *voxel_search( );
    void readscene( );
    void Setup_Octree( );

    tree = 1;

    /* Get the Kernel Linda Environment */
    Val env = environment( );

    /* Read scene file name from tuple space */
    env.rd("SCENEFILE",scenefile);

    /* Read the scene database */
    readscene((char*)scenefile);

    /* Setup the octree for the scene */
    Setup_Octree( );

    if(Root->Volume == EMPTY_With_Children)
    {
        root = voxel_search(Root,ORIGIN);
        while(root->Volume == EMPTY_With_Children)
            root = voxel_search(root,ORIGIN);
    }
    else
    {
        root = Root;
    }

    int *scanline = (int *) malloc(3*X_res*sizeof(int));

    while(env["Root"]["DICTIONARY"]["SCANLINE",i].inp("SCANLINE",i))
    {
        AXIS.x = Screen.x;
        AXIS.y = Screen.y + (float)i ;
    }
AXIS.z = Screen.z;

while(AXIS.x < (Screen.x + (float)X_res)) {
    pixel = AXIS.x - Screen.x;
    Pixel_Intensity.r = 0.0;
    Pixel_Intensity.g = 0.0;
    Pixel_Intensity.b = 0.0;

    DIRECTION.x = AXIS.x - ORIGIN.x;
    DIRECTION.y = AXIS.y - ORIGIN.y;
    DIRECTION.z = AXIS.z - ORIGIN.z;

    DIRECTION = normalize(DIRECTION);

    Pixel_Intensity = tree_search(root, flag, tree, ORIGIN, DIRECTION);
    Pixel_Intensity.r = Pixel_Intensity.r * 255.0;
    Pixel_Intensity.g = Pixel_Intensity.g * 255.0;
    Pixel_Intensity.b = Pixel_Intensity.b * 255.0;

    scanline[((int)pixel*3)] = (int)Pixel_Intensity.r;
    scanline[((int)pixel*3)+1] = (int)Pixel_Intensity.g;
    scanline[((int)pixel*3)+2] = (int)Pixel_Intensity.b;

    AXIS.x += 1.0;
}

printf("scanline %d ", (int)i);fflush(stdout);
block = createblock((int)X_res*3*sizeof(int), (char*)scanline);

env["Root"]["DICTIONARY"]["MEMORY"].out(i,block);

}